SUBMICRON SYSTEMS ARCHITECTURE
SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

5122:TR:84

Computer Science Department
California Institute of Technology

March 1984

SUBMICRON SYSTEMS ARCHITECTURE

SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

5122:TR:84

Computer Science
California Institute of Technology

March 1984

Submicron Systems Architecture

Semiannual Technical Report

Computer Science
California Institute of Technology

5122:TR:84

March 1984

Reporting Period:	16 October 1983 to 15 March 1984 (5 months)
Principal Investigator:	Charles L Seitz
Faculty Investigators:	Randal E Bryant James T Kajiya Alain J Martin Robert J McEliece Martin Rem Charles L Seitz

Sponsored by the
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

SUBMICRON SYSTEMS ARCHITECTURE

Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document reports the research activities and results for the five month period 16 October 1983 to 15 March 1984 under the Defense Advanced Research Project Agency (ARPA) Submicron Systems Architecture Project.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes, and includes related efforts in concurrent computation and VLSI design. Additional background information can be found in previous semiannual technical reports [5052:TR:82, 5078:TR:83, 5103:TR:83].

1.3 Highlights

The highlights of the previous 5 months are:

(1) The cosmic cube has been operating very reliably. Its performance has been improved, and system software development is proceeding well and on schedule (section 2.1).

(2) The layout of the mosaic RAM is complete (section 2.2.1).

(3) A concurrent circuit simulation program is now working in part, with completion of this effort expected in June (section 3.2).

(4) The design and simulation of the Mossim Simulation Engine (MSE) is complete, and this special purpose system is under construction (section 4.1.1).

(5) A new architecture for Reed-Solomon decoders based on bit-serial multiplication over finite fields has been developed (section 4.2.1).

2. ARCHITECTURAL EXPERIMENTS

The Submicron Systems Architecture project has three architectural experiments, Cosmic Cube, Mosaic, and Super Mesh, in various phases of design, construction, programming, and use. These machines are all ensembles of identical, concurrently operating, and regularly interconnected elements that communicate by message passing. Our priority in these efforts has been to apply VLSI technology to achieve substantial advances in cost/performance in a limited set of computationally demanding problems.

A "position paper" on these experiments, which describes in detail the principles and design issues for this family of architectures, the current status of the experimental machines, and the system and application software, is included at the end of this report. The following sections summarize the activities and progress in past five months.

2.1 Cosmic Cube

(W C Athas, Reese Fawcette, Chuck Seitz)

2.1.1 Hardware Status

(Bill Athas, Chuck Seitz)

The 64 node Boolean 6-cube machine has been running very reliably, with over 95% utilization, for the past 5 months. It has experienced one hardware failure, an intermittent failure in one 64K dRAM chip detected by a high incidence of parity errors in this node, and which was fixed by replacing the chip. The 8 node (3-cube) machine has not experienced any failures. This one failure in 230,000 node-hours is consistent with the calculated MTBF of about 100,000 hours, and in fact implies that the calculated MTBF is conservative at an 80% confidence level.

Approximately 20 errors have been detected in the intermediate host to node 0 channel due to crosstalk in the cable, a problem that is being fixed by a new cabling arrangement and improved versions of the CLS (cube life support) board in the intermediate host and of the "corner" node (see 2.1.2 below). There have also been about 20 soft errors (detected by parity) in the 8 Mbytes of dynamic RAM.

With the receipt of sufficient 5 MHz versions of the Intel 8087 floating point coprocessors, the clock rate of the nodes in the cosmic 6-cube and 3-cube machines were increased from 4.1 MHz to 5.0 MHz without any problems. Except for the 8087's, the system operates reliably at up to 7.7 MHz. Our current benchmarks can therefore be expected to improve by about 50% when 8 MHz 8087 parts become available.

An SU(3) lattice gauge theory computation programmed by Steve Otto, a physics postdoctoral fellow, has been run for over 2,000 cumulative hours on the 6-cube, with about 10 hours between checkpointing. Since this program uses all of the storage of each node and exercises all of the node

functions, it is an excellent test of the 6-cube. This program, which benchmarks at about 10 times the speed of a sequential version running on a VAX11/780, has also produced some results that our physics colleagues regard as very significant. There are now several other application programs running that were developed by people in other groups. These programs exhibit results in (FFT) Fourier analysis and matrix computations that are very close to performance modeling predictions.

2.1.2 Hardware Improvements

(Bill Athas, Chuck Seitz)

The overall hardware configuration for the cosmic cube includes mainframe computers such as a VAX for program development, which is coupled to the "intermediate host" (IH) computer, which in turn is coupled to one or more of the cube corner boards. The intermediate host is a microprocessor based system using the multibus backplane. The use of multibus allows the link from mainframe to intermediate host to be an Ethernet connection that can be assembled with off the shelf parts. Software drivers for the IH are readily available from the department's Ethernet project. The link from intermediate host to the cube corner board requires a special interface. This interface requires just one card slot of the multibus backplane, and is called the Cube Life Support (CLS) board. The CLS currently in use provides six standard cube channels, clock, RAM refresh interrupt, global line interfaces, and some other miscellaneous functions.

We decided to construct an improved CLS board, called "Rosebud", to serve as the archetype with minor variations for different implementations of this interface function, for two reasons: First, the interface required for mosaic systems is very similar. Second, we were experiencing a very small (less than 1 error per 10^{12} packets) but annoying error rate traced to crosstalk in the ribbon cable that connects the CLS to the corner node, which required some rebuilding in any case. Extensive use of commercial PAL chips allow this design to be adapted between different versions of Rosebud only by changing PALs, ROMs, and the communication channel logic that is idiosyncratic to the particular machines.

While PALs have been used occasionally in previous projects, this application and the MSE project use PALs so extensively that it was worthwhile to improve the tools that we use to program PALs. In particular, a new version of the program that converts state tables into Boolean switching expressions can handle larger and incompletely specified state tables. This program may prove useful also for chip designs.

2.1.3 System Software Status

(Bill Athas, Reese Fawcette, Chuck Seitz)

Last summer a resident operating system to run in each node of the cosmic cube was specified and designed. This "cosmic kernel" [5095:TR:83] supports multiple processes per node element, message routing, and queueing of messages in transit. (See our previous technical report [5103:TR:83 for a

more complete description.) From this detailed specification, coding of the operating system in 8086 assembly code was started. The operating system is on schedule for first use in applications in April. This date is coordinated with the schedule for porting the circuit simulator (see section 3.1.2) onto the cosmic cube. The circuit simulator is a good test case for the operating system. Another test program, written by Craig Steele and Bill Athas, is a neuron simulation program based on Hopfield's neuron model. Both the circuit simulator and the neuron program are written in Pascal, and depend on an 8086 Pascal compiler running on our VMS VAX.

Here are some details of the "cosmic kernel" coding effort:

The procedures, data structures, and algorithms for the CK have all been worked out and are mostly implemented. Development is currently being done on a single 8086 with a terminal line and a simple monitor. The code for the system is highly parameterized, so that many facets of the system can be changed with just an equate change and re-assembling. This parameterization includes such items as total memory size, number of processes permitted per processor, all table sizes, etc. Other parameters, such as the dimension of the cube are determined dynamically by our present initialization and bootstrap system [5070:DF:83].

Much of the work has gone into devising an acceptable method of queueing and otherwise dealing with incoming and outgoing messages. The current system reads in the header to a message and then determines if the target process is resident in this node. If it is not, the message is read and buffered, then entered into the send queue of the appropriate outgoing channel. If the target is resident, a check is made to see if a RECV has been executed that will accept a message of this type. If yes, the message is not buffered, but copied directly into the target process's data space. If no RECV is pending, then the message is queued until such a RECV is executed. When a SEND is executed, the message is queued for transmission, but remains in process data space until it is sent out over the channel. At this time, the lock on the buffer is reset indicating message sent. Messages between processes in the same node are handled somewhat differently in that the SENDing process does not have the lock reset on its buffer until the RECV is executed by the target process. The message never enters queue space.

The memory management system is also worked out and coded, although it may need to be optimized if it is observed that compactions of memory are done more frequently than expected. Currently, all memory not used by the CK is available for use by user processes or as queue space for messages. There is a marker above which all memory is used for process space, and all below is used for queue space. If there is a request for more space in a section than is available in that section, the marker is moved up or down accordingly and the space changes hands between the two possible uses.

The code is intended to be sufficiently modular that any given function may be re-written entirely without requiring any changes elsewhere in the system code. This can easily be done with scheduling algorithms, I/O handling, memory management and other sections that may be interesting to experiment with. The interactions between separate functions such as I/O and memory management (an example of such interaction is when input on a

channel is blocked because of insufficient queue space) is all handled through very general flags and pointers in globally defined tables.

Since the process spawner is itself intended to be an Outer Kernel process, getting it loaded into the machine requires some finesse. The solution used is to load it at the same time as the Inner Kernel is loaded, using the bootstrap program resident in eprom on the node boards. The spanwer uses privileged system calls to create a process table entry for the new process, and will take care of the communication with the intermediate host necessary to initiate the loading of the new process.

2.1.4 Advanced Technology Homogeneous Machines

Design of an advanced technology node for a homogeneous machine with characteristics described in our previous technical report [5103:TR:83] is proceeding.

2.2 Mosaic Systems

(Chris Lutz, Steve Rabin, Don Speck, Francoise LeMouel, Chuck Seitz)

For background on this project, please see: (1) A paper on the design of the Mosaic element from the Proceedings of the MIT Conference on Advanced Research in VLSI, January 1984, included as an appendix of our previous technical report [5103:TR:83]; (2) the discussion of the mosaic project in the paper that is included as an appendix to this report.

Activities in algorithms and programming systems for Mosaic are deferred to the concurrent computation (section 3) below.

Following the large "bursts" of design and testing effort reported in the two previous semiannual technical reports, the progress on the mosaic designs has been relatively slow in this five month period. The decision to install Berkeley UNIX 4.2 bsd immediately on receipt of the distribution tape, modulo bugs, brought those design activities that required use of our VAX to a halt during the first two months covered in this report. Even after the new operating system was running, we had to modify existing design tools to function correctly under 4.2 bsd.

Nevertheless, there has been substantial progress in design and in design improvements, notably (1) the completion of the mosaic RAM layout (2) testing by simulation and eliminating bugs in the latest (third) generation of the microcode, (3) in the logic design of a new mosaic communication section, (4) testing logistics, and (5) yield evaluations of mosaic parts in preparation for fabrication of mosaic node elements with on-chip RAM. We have also been examining with ISI the interesting possibility of constructing mosaic elements on ceramic hybrid packages using a new MOSIS service. This approach would allow smaller, high yield die to be used to assemble mosaic elements with less sensitivity to yield problems, and potentially with much more storage.

2.2.1 Mosaic RAM

(Steve Rabin, Chuck Seitz)

A complete layout for the Mosaic RAM described in the previous technical report [5103:TR:83] is now complete. A 4K bit (1 64x64 bank) prototype is being fabricated on the 22 March 84 MOSIS 3 micron nMOS run. The accompanying diagram, figure 1, is a floorplan of a 16K bit (4 bank) memory, indicating the composition of the various components of the memory. The Bus, Decode and Word Select (W.S.) components are shared between adjacent half-banks, with the proviso that Word Select and Data Bus tap components can only be shared by banks Hamming distance 1 apart. For a more detailed (confusing?) description of the pipelining being used in this memory design, please refer to [5093:TR:83].

2.2.2 Mosaic Microcode

(Chris Lutz, Chuck Seitz)

Taking further advantage of the modifications to the Mosaic processor that allowed interrupt handling, the design has been augmented to include a "soft reset". This improvement allows on soft reset complete recovery of the state of a mosaic ensemble. The ensemble can be restarted later with the same state. This feature can be used as a diagnostic aid, to allow periodic backup of long-running tasks, or to swap tasks and thus allow time-sharing of the ensemble.

The software floating point routines have been updated to take full advantage of recent instruction set changes. Most notably, the microcoded multiply instruction is used to perform a floating multiply in less than a quarter the time required by previous software. This improvements will double the processor's performance at tasks which are largely floating point.

Additional verification efforts on this third edition of the microcode discovered a couple of bugs that were fixed. The attached appendix B documents the complete mosaic instruction set and microcode.

2.2.3 New Mosaic Ports

(Francoise LeMouel, Chuck Seitz)

A new set of ports of the mosaic element are being designed, using a memory-mapped rather than instruction driven approach, and using phase-insensitive rather than synchronous communication between chips. This modular design will be interchangeable with mosaic RAM sections, and will thus allow mosaic elements to be configured with variable numbers of ports, and systems to be configured with communication plans of degree larger than 4, such as Boolean n-cubes, and without concern about clock skew. Logic design is complete; layout is in progress. The circuit style is the same "hot-clock" nMOS as other mosaic parts.

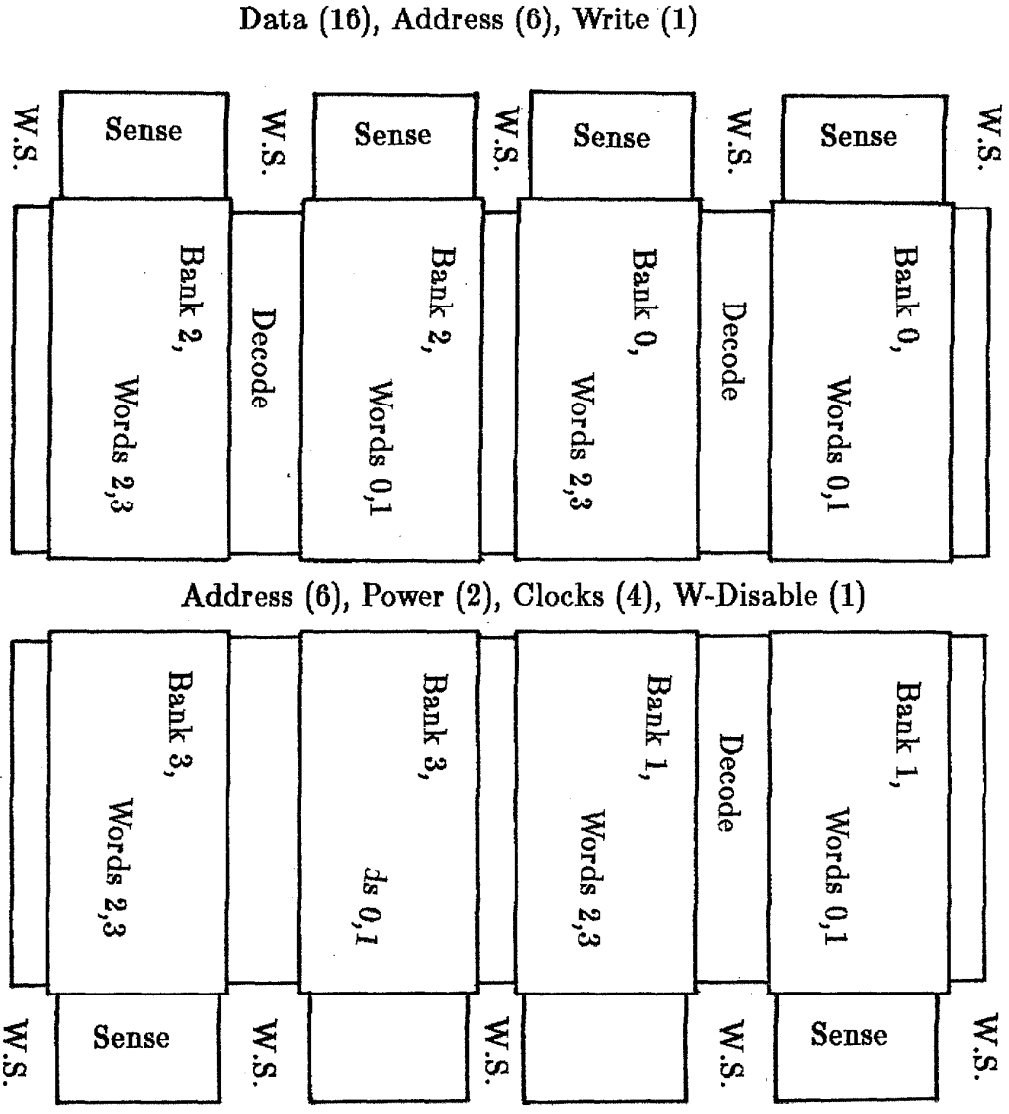

Data (16), Address (6), Write (1)
W.S.
Sense
Bank 2,
Words 2,3
Decode
Bank 2,
Words 0,1
Bank 0,
Words 2,3
Decode
Bank 0,
Words 0,1
Address (6), Power (2), Clocks (4), W-Disable (1)
Bank 3,
Words 2,3
Bank 3,
ds 0,1
Bank 1,
Words 2,3
Decode
Bank 1,
Words 0,1
Sense
Data (16), Address (6), Write (1)

Figure 1

2.2.4 Mosaic Yield Evaluation

(Steve Rabin, Don Speck, Chuck Seitz)

Some modifications to the 10,000 lines of Earl code that define the layout of the mosaic processor prototype with padframe now allow this chip to scale down to a 2.5 micron feature size (lambda = 1.25 microns). Mosaic processor prototypes feature sizes of 2.5, 3.0, 4.0 and 5.0 micron (lambda of 1.25 to 2.5 micron) have been submitted to MOSIS as process evaluation test structures (technology: PCM).

2.3 Super Mesh

(Wen King Su, Chuck Seitz)

Super mesh is a serial communication, serial floating point arithmetic, SIMD machine in the middle stages of design. Its design is discussed in detail in the attached appendix A.

A super mesh chip is being laid out by Wen-King Su and a team of students recruited from the Caltech VLSI Design Laboratory course. The layout is progressing rapidly, and is expected to be complete by June.

3. CONCURRENT COMPUTATION

3.1 Cosmic Cube C Optimizer

(Mike Newton, Chuck Seitz)

A code optimizer to improve the 8086/8087 code produced by our C compiler is now in regular use by everyone using the programming language C on the Cosmic Cube. The optimizer reduces the lines of assembly code produced by the C compiler by approximately 35%, and reduces the code space by approximately 20%. Several of the major optimizations are in the control of loops, so run time improvement is expected to be in the 30% range, but this has not yet been verified.

The project that inspired the optimizer, a version of concurrent chess, has not been doing so well. Largely due to the space limitations the compiler and linker have yet to be able to successfully process the code. It is hoped that further additions to the optimizer and further reductions to the chess program will solve this problem.

3.2 Circuit Simulation

(Sven Mattisson, Chuck Seitz)

A study of concurrency opportunities in circuit simulation, reported in our previous technical report [5103:TR:83] examined suitable algorithms and formulation methods for implementing a circuit simulator on a multiprocessor system. Based on that study, a program for the cosmic cube is being developed. Circuit simulation can be partitioned to processes of sufficiently fine grain that this formulation could be used in the future for a circuit simulator to run on a mosaic system.

The nodal formulation method is chosen as network description, because it preserves the sparse connectivity properties of the circuit and has a tendency towards diagonal dominance for MOS circuits. However, this formulation method precludes the use of current controlled branches and ungrounded voltage sources. However, when simulating real circuits this is not a serious drawback, since almost all device models describe branch currents as functions of branch (node) voltages.

In order to solve the matrix equation resulting from the nodal formulation concurrently, the Jacobi method is used. This method allows the matrix equation to be partitioned into row equations that can be solved concurrently, with only node voltages being exchanged between different processors. Also several rows can be clustered into one process, requiring only "external" node voltages to be imported and externally used "internal" nodes to be exported to the process.

Within each process, the row equations are solved iteratively with modified Newton-Raphson iterations as inner iterations. The use of modified Newton-Raphson iterations enables numerically efficient approximate formulas for the derivatives of the branch currents to be used. Furthermore,

the derivatives need not be evaluated every N-R iteration. The accuracy of the final result is determined only by the accuracy of the branch current equations.

To take advantage of latent nodes, a time window integration scheme is used. This decision implies that each node voltage (or cluster of nodes) is solved for a part of the waveform, and that those parts, for all nodes, are iterated. When the waveforms for the time window have converged, a new time window is selected, and the iterations are repeated. With this scheme waveforms corresponding to nodes that do not change their voltage allows large steps in the integration formulas to be used, and thus enhance computational speed. To take advantage of this latency stiff backward differentiation formulas are used to integrate the waveforms.

A concurrent circuit simulation program has been developed to evaluate the approach outlined above. This program is written so that it runs also on a VAX, but can be ported to the cosmic cube with minimal change. So far the generation of the linked list data structure is complete. The DC-analysis of nonlinear (MOS transistors and diodes) is also done and works (produces the same answers as SPICE with the same device models). However, to guarantee convergence the DC-analysis really has to be done as a transient analysis with constant sources. The transient analysis part is presently under development. After completion of this last part, experiments on the cosmic cube can start.

The data structure is designed such that each row is independent of every other, enabling total freedom in partitioning. Thus associated with each row is a list of all devices connected to that row, and only node voltages for nodes with a branch connected to the row in question need to be known. Since only some of the equations for a device enters in the network equations for each row, it is connected only to those model equations necessary to calculate branch currents and conductances, in order to reduce storage overhead caused by the multiple entries of the device.

So far everything is going very well, and we are somewhat ahead of our target of having a concurrent circuit simulator running on the cosmic cube by June.

3.3 Process Placement

(Craig Steele, Chuck Seitz)

Craig Steele has been working on optimization of process placement on distributed homogeneous processor architectures. In a such an environment, a computation may be represented as a graph, where communicating processes are the vertices, and logical communication channels are the edges. Likewise the underlying physical structure may be represented as a graph with nodes and physical communication paths represented as vertices and edges. To run a computation on a distributed processor, the processes must be loaded onto the physical machine, requiring a mapping of the logical graph onto the physical graph.

Minimizing the communication cost of the computation is a major problem unless the logical and physical graphs are similar. While many problems of interest to physicists can take advantage of the isomorphism of the 6-cube architecture of the Cube to three-space, even in three-dimensional simulations the density of simulated objects may vary, lowering efficiency with simple partitioning. Graph structures common in applications of interest in computer science, such as trees, do not map in obviously good ways to n-cube architectures.

Using the technique of simulated annealing, good mappings may be found in moderate time for arbitrary logical computation graphs. Taken into account are arbitrary edge weights (reflecting varied utilization of logical communication channels) and arbitrary process memory sizes (allowing the physical processor constraints to affect the density of processes per processor).

Work is continuing in using these mapping techniques to compare various communication structures for homogeneous machines, and in discovering good ways to compute these mappings on a concurrent architecture.

3.4 Tree Machine Software

(Pey-yun Peggy Li, Lennart Johnsson)

A mapping algorithm has been developed to map a non-binary, oversized logical tree onto the fixed sized binary tree machine. The mapping result preserved the balance and the regularity of the logical tree. When the height of the logical tree is larger than or equal to that of the binary tree, the logical tree is directly mapped onto the binary tree. Hence, the adjacent nodes in the logical tree are either located in the adjacent processors or shared in one single processor. While the height of the logical tree is smaller, the padding nodes are inserted into upper levels of the logical tree to take advantage of all the available nodes in the binary tree.

The mapping algorithm can be embedded into the downloader with logical tree construct as input. It runs recursively and distributively in the tree machine. No extra memory space is required as long as the maximum number of nodes shared in each processor doesn't exceed its memory limitation. No global address information is needed by the host or the tree processors, such as the size of the logical tree and the tree machine, the global address for each tree processor, etc. The complexity of the mapping algorithm is roughly equal to that of the downloader, namely, at best O(log N), where N is the total number of nodes in the binary tree.

The scheduler has been modified to be capable of addressing the non-binary fanouts of each process. Unified I/O macros are introduced in the user program, which will select the proper I/O routines according to the mode of the processor. This capability allows the mixture of the single node processors and time-sharing mode processors in the tree machine. Furthermore, how the machine runs is totally transparent to the user.

3.5 Communication Primitives for Concurrent Computation

(Alain Martin)

We consider a programming language kernel for a distributed computation consisting of a sequential part -- with assignment, alternative command, repetition, recursion, and sequential composition -- extended with the concurrent composition of processes and communication primitives. (CSP and its variants are such a kernel.)

The notion of suspension of a communication action is essential to describe the progress and fairness of a computation. Yet, none of the currently used communication primitive sets makes it possible to determine directly the suspended state of an action.

We propose a simple addition to the communication primitives used in such a kernel. This extra Boolean primitive, called the probe, makes it possible to test whether a communication action is pending on a channel.

Be separating the testing of a pending communication action from the "firing" of the communication, the semantics of communication are both simpler and more general. The communication commands are entirely symmetrical in input and output. Yet we do not encounter the semantic and implementation problems occurring in CSP-like languages when input and output are allowed in guards.

The negated probe considerably enhances the possibilities of the language by making the implementation of fairness properties and "timer-like" constructs possible and easy.

Finally, the implementation of the probe poses no major problem. (A PROBE call has been included in the cosmic kernel (see 2.1.3).) The results of this research are reported in [5124:TR:84].

3.6 Modic Compiler

(Blake Lewis, Alain Martin)

As previously reported, a Modic compiler for mosaic systems is being designed and implemented. The kernel of the language has been fixed, and the generation of an LL1 parser is well underway.

3.7 Functional Programming

(Jan L A van de Snepscheut)

We are experimenting with functional programming languages to find out whether they are suitable for expressing concurrent computations. From the programmer's point of view it turns out to be very convenient to allow infinite data structures as intermediate results. A functional program is then composed from two classes of functions: one class of functions that define infinite data structures, and one class for selecting some finite

substructure thereof. We are working on an implementation method in which the actions to be performed are drived from the first class of functions and the order in which they are performed from the second class. We are beginning to suspect that, in a distributed implementation, in addition to local storage some form of global storage is desired.

3.8 Signal Processing and Image Analysis

(Leonid Rudin, Jim Kajiya)

Leonid Rudin has been working on the following research problems in the field of theoretical signal processing and image analysis:

(1) Development of new nonlinear approach to image enhancement with simultaneous smoothing that would not produce the spurious ringing effect which can be shown to be an artifact of the traditional linear filtering methods.

(2) Investigation into the ill-posed nature (in the sense of Hadamard) of image processing tasks from the standpoint of differential equations theory rather than as commonly described in the signal processing literature approach which uses first kind Fredholm equations. This direction seems to be very promising, and has already allowed preliminary theoretical calculations for the case of Gaussian kernels (images blured by atmospheric turbulence).

(3) In connection with (2), a new class of image processing techniques seems to be evolving, which we will call partial restoration backwards in time (space). In this approach fast algorithms of linear programming will be playing a major role.

(4) Foundations of homomorphic signal processing which should lead to the better theoretical understanding of this unique non- linear technique and aid in developing an effective and less restrictive phase unwrapping algorithm.

(5) These later developments do suggest that the traditional linear, space invariant systems approach is a very limited view of the proper mathematical models for signals and signal processing systems.

(This research is only partially supported by ARPA.)

4. VLSI DESIGN

4.1 Switch Simulation Tools

4.1.1 The MOSSIM Simulation Engine (MSE)

(Bill Dally, Randy Bryant)

Design verification is essential in the development of VLSI systems. The complexity of VLSI circuits, inaccessability of internal nodes, and difficulty of repair make the probability of producing a working chip very low without extensive design verification. As the complexity of VLSI circuits approaches 10^6 devices, the computational requirements of design verification are exceeding the capacity of general purpose computers. To provide the computing power required to verify these complex VLSI chips, we are developing the Mossim Simulation Engine (MSE) [5100:DF:83].

The Mossim Simulation Engine is a special purpose processor which, in a single processor configuration, performs switch-level simulation of MOS VLSI circuits 200-500 times faster than a VAX 11/780. In multiple processor configurations even greater speedup can be achieved.

The MSE overcomes two limitations of existing simulation engines. By using the switch-level model developed by Bryant [5065:TR:83], the MSE performs accurate simulation of MOS circuits. Existing simulation engines perform logic simulation and cannot model MOS effects such as stored charge, charge sharing and transistor ratios. Also, by using the concept of virtual processors the MSE can simulate a circuit many times larger than the size of the processor. Existing simulation engines are limited to simulating circuits which fit in the processor.

The design of the MSE is now complete and has been simulated at the chip level using the DSIM simulation system. We are currently working on construction a wire-wrap prototype of a single processor MSE. This prototype will be used both as a research tool to support the simulation requirements of the next generation of VLSI circuits and as a test bed for experiments in switch-level simulation including: the development of a virtual simulation processor system, experiments in the application of multi-processing to switch-level simulation, and a study of the locality of activity in MOS circuits.

4.1.2 The DSIM Functional Simulator

(Bill Dally, Randy Bryant)

The DSIM functional simulation system [5109:DF:83] consists of an event driven timing simulator, DSIM, a modeling language and compiler, DCOMP, and software to support physical design, DPHYS. DSIM simulates a digital system described as a connection of functional modules ranging in complexity from logic gates to complete microprocessors. The DSIM timing primitives can be used to model absolute delay, ambiguity delay, and inertial delay.

Functional models are programmed in an extension of the Mainsail programming language and are compiled using the DCOMP compiler. The physical design software produces wire lists and can interface with automatic wire-wrap machines.

The DSIM system is technology independent. It has been used extensively in the development of the Mossim Simulation Engine (TTL/Wire-Wrap technology), and is also being used by several ongoing VLSI projects using nMOS and CMOS technologies.

4.1.3 HEX, a Hierarchical Extractor

(Yen-jen Oyang, Randy Bryant)

HEX, a hierarchical circuit extractor, is implemented. HEX exploits the hierarchy in CIF files to avoid redundant analysis work. The speedup depends on the regularity of chips. In the best cases, HEX runs about 10 times faster than Tom Hedges's extractor, which is a flat circuit extractor. HEX generates hierarchical output format to describe the circuit denoted by CIF files. HEX is expected to become available in April, 1984.

4.2 Error Correction

4.2.1 Bit-serial Multiplication over Finite Fields

(Doug Whiting, Bob McEliece)

Bit-serial multiplication over finite fields has recently been given much attention in the literature. Berlekamp has shown how to build efficient Reed-Solomon encoders using a dual basis bit-serial multiplier, and implementations of his technique have been realized in both conventional logic and VLSI chips. Upon contemplating this breakthrough, it becomes apparent that the major conceptual advance is in the bit-serial multiplication method; the encoder is but one particular application. Although Berlekamp applied this technique only to multiplication by known constants, it can readily be generalized to the product of two arbitrary field elements. Omura and Massey have presented another scheme of performing bit-serial multiplication, involving the use of a basis consisting of all conjugates of a given field element. The question then naturally arises: which of these approaches is more efficient? To our knowledge, no one has yet addressed the issue of choosing the representation of the field elements which (in some sense) minimizes the cost of building a bit-serial multiplier.

Our research has investigated the structure of finite fields to see if a basis suitable for efficient bit-serial multiplication can be found. The result is a family of bit-serial multipliers which includes Berlekamp's and Omura's approaches as special cases; among all these methods, the dual basis structure is shown to be superior in several practical respects. Further, it has been proved that a self-dual basis can be found if and only if some elements of the field have a trinomial as their minimal polynomial,

which result unfortunately excludes GF(256). We have also demonstrated how multiplicative inversion can be accomplished in a bit-serial fashion, greatly reducing the complexity traditionally associated with this operation.

Our initial goal was to investigate the design of single-chip Reed-Solomon decoders. In the past, the complexity of a decoding algorithm has often been measured by the number of finite field multiplications required, because the design of high performance decoding systems has typically revolved around a single, extremely fast, fully parallel multiplier. While it is quite feasible to build a parallel multiplier on a VLSI chip, the structures involved are rather unwieldy. Multiplication of two elements of $GF(2^m)$ requires m^2 Boolean AND operations, together with at least m^2 exclusive-or operations. From wiring considerations alone it is clear that at most a few such multipliers could fit on a single chip, for the fields of interest.

Multiplication over $GF(2^m)$ has an area-time complexity of m^2, and parallel multipliers achieve this with area proportional to m^2 in one unit of time. We hope to be able to make an area-time tradeoff by building multipliers which require area proportional to m and produce the entire product in m time units. But note that it is not altogether clear which approach is faster in VLSI, since the unit of time for a parallel multiplier may well be considerably longer than that of a bit-serial multiplier! Employing bit-serial techniques present the possibility of fitting large numbers of multipliers on a single chip, molding the chip architecture directly to the decoding algorithm of interest.

In VLSI, the complexity of an algorithm is often determined more by communication flow than by an overall count of computational operations. Using a small number of parallel multipliers in a decoder chip is inefficient for two reasons. First, each multiplier will typically be multiplexed between many inputs and/or outputs, and the cost of such sharing (as measured in chip area) may well outweigh the cost of the multiplier itself! Second, and perhaps more important, is the fact that the polynomial manipulations involved in decoding algorithms have a high degree of inherent parallelism. By executing many multiplications concurrently, even if each operation is (individually) relatively slow, large gains in throughput can be achieved.

Currently we are investigating the structure of various decoding algorithms to see how to map them onto a single chip, using the bit-serial arithmetic techniques which have been developed. A group from the Caltech VLSI Design Laboratory class is working on implementing the Berlekamp-Massey algorithm on a single chip. Many similarities between algorithms are made manifest when they are examined with an eye to parallelism and VLSI implementation. This comparison between algorithms is being prepared and will be included as part of Whiting's PhD thesis.

4.2.2 Soft Error Correction for Increased Densities in VLSI Memories

(Khaled Abdel-Ghaffar, Bob McEliece)

If VLSI RAM densities are to continue to increase, it will undoubtedly be necessary to take the problems associated with soft errors much more seriously than has previously been done. We have proposed a methodology for analyzing the effects of soft errors in VLSI RAMS as feature sizes decrease, and for taking corrective action with error-correcting codes. We have taken a parametric approach, making several different assumptions about how the error severity will scale as feature size decreases, and our conclusions are stated relative to the particular assumption made.

The most significant source of soft errors in the near future is expected to be alpha-particle effects. The primary source of these alpha particles appears to be spontaneous emission from trace impurities in the IC package. The charge track produced in the memory cells by a sufficiently energetic alpha particle can change the logical state. Cosmic rays can produce errors by essentially the same mechanism.

If feature size becomes sufficiently small, thermal and quantum effects may become significant, and we have shown, will impose an ultimate limit on information densities.

We have shown that error-correcting codes can be used to reduce the effects of soft errors on dense memories. Unfortunately, however, no accurate models have been established which predict the severity of error at submicron feature sizes. Thus, as mentioned above, we have pursued our study using a family of abstract error models. We have studied the improvement possible (as measured in area per information bit) with coding, for each of our abstract models. We have also considered the problem of placing the encoder and decoder on the RAM chips. We have shown that an explicit class of codes, orthogonal codes, are attractive for a certain class of error models.

4.3 Self-Timed Systems

(Jan L A van de Snepscheut)

We are exploring the possibilities of characterizing self-timed systems as systems that exclude interference. Traces and trace structures are used for proving the absence of interference in certain compositions of components. Such composition rules allow hierarchical structuring of self-timed systems.

4.4 A Hierarchical General Interconnect Tool

(John Ngai, Chuck Seitz)

SMART (Simple Minded Approach Routing Tool) is a general interconnect tool for large-scale MOS custom integrated circuits. The initial prototype implementation became first operational in January, 1984, and is now being

tried out on the VLSI Design Laboratory class. The system contains knowledge about geometric design rules. It supports both single layer metal nMOS and CMOS/SOS processes, although future versions will support also CMOS/bulk and multi-layer metal processes.

SMART accepts a hierarchical declaration of routing contexts in terms of modules. A module can be as simple as a leaf cell or as complicated as an entire IC chip. Through hierarchical nesting of modules, SMART allows composition of large complicated circuits in terms of simpler ones.

SMART supports a fairly general planar routing of single layer metal power and ground busses. Widths of power busses are determined explicitly by the user in a hierarchical form. Optimization of metal layer usage is done in place during the actual wire routing phase, and consequently requires much less CPU time than separate post processing. Both the utility and the performance of this program seem quite promising.

A user manual of this program is available as a Caltech technical report [5118:TR:84]. A detail report on the algorithms used in this program is in preparation.

4.5 CAD Tools

The release by Berkeley of UNIX bsd 4.2, and the decision to immediately install this new operating system (modulo bugs) brought most design activities to a halt during the first two months covered in this report. Subsequently local design tools have been modified to function correctly under the new operating system.

Other modifications to the design tools include support for bulk CMOS and PCB technologies.

4.6 Hot-clock nMOS

4.6.1 A Hot-clocked nMOS Multiplier Project

(Sandy Frey)

As technology changes and our understanding of it increases, the designs that best implement standard functions may also change. Using pass transistors is a particularly efficient way to implement logic functions in nMOS technology. In this project, a multiplier/accumulator has been designed that replaces the standard binary carry save adder tree, used in many multipliers for summing partial products, with a quaternary adder tree design chosen to make more efficient use of pass transistors.

A second novel feature of the multiplier/accumulator design is the "hot-clock" driving technique used to restore signal levels and to enhance the performance of the pass transistor circuits. Use of these drivers eliminates the need for depletion mode transistors. The two non-overlapping clock signals that drive these circuits are obtained from off the chip. They

are also used to provide power and ground throughout the chip, eliminating the need to distribute those signals.

The multiplier/accumulator is pipelined to increase throughput. Two non-overlapping clock pulses are used every clock cycle. The function is subdivided into the following six pipelined stages operating on successive clock phases: (1) Booth Algorithm, (2) Multiplication, (3) First Stage Carry Save Adder, (4) Second Stage Carry Save Adder, (5) Carry Lookahead Adder, (6) Output to pads. These six clock pulses correspond to a latency between input and output of three clock cycles with a new multiplication starting every clock cycle.

Many SPICE simulations have been run in designing the multiplier/accumulator. They primarily have been used to obtain a balance in the estimated delays of the different portions of the pipeline implementation. The goal of the project is to accept a 50 nanosecond clock cycle. Thus each portion of the pipeline must operate in 25 nanoseconds. Analysis by SPICE simulation has shown that each pipeline segment should just barely accomplish this.

These simulation results are based on estimates of parasitic capacitances and a benign environment. It will be pleasing to everyone if the measured performance of the finished chip in 4 micron nMOS with single level metal, is within a factor of two (i.e. 100 nsec clock cycle) of the simulated results.

4.6.2 Hot Clock Design Style

(Chuck Seitz, Sandy Frey, Sven Mattisson, Steve Rabin)

Use of "enhancement-isolation clock-and" circuits in the prototype mosaic processor and ports has been described in previous technical reports and in [5093:TR:83]. Test structures submitted on 22 different MOSIS nMOS runs have been tested and their behavior agrees with that predicted by circuit models. The simpler form of these "hot clock" bootstrap drivers have also been used in numerous mosaic runs without problems.

The multiplier project described above, and the mosaic RAM design, are extending this design style to an extreme point in which depletion transistors are entirely absent, hot-clock circuits are used to perform logic operations, and in which VDD and GND are seldom used. This extension is being pursued not just for the sake of exploring an interesting extreme, but because it is a way to achieve excellent performance in very low power nMOS designs. Indeed, this technique allows hot clock nMOS circuits to operate on much less power than CMOS designs, because the dynamic power normally dissipated on the chip by driver circuits changing the voltage on capacitances (CV^2f power) is instead "reflected" to off-chip clock driver circuits. Because these off-chip clock driver circuits can use inductances, it is possible to circumvent almost all of the dynamic power dissipation.

For example, a 4K bit mosaic dynamic RAM section "reflects" about 0.2 watts of clock power (that is, 0.2 watt would necessarily be dissipated in the source impedance of the clock driver at 10 MHz if the source impedance were

real), but dissipates less than 0.02 watts. These circuit energetics are sensitive to, for example, the clock slope.

A paper on hot-clock nMOS is in preparation, and will be included in the next semi-annual technical report.

ARPA Bibliography

"Experiment..." paper

<end>

California Institute of Technology
Computer Science, 256-80
Pasadena, California 91125

ARPA Technical Memos and Technical Reports
together with selected other Caltech reports on VLSI topics
March 1984

Available from the Computer Science Department Library
+++

5125:TR:84 "Super Mesh," MS Thesis, March 1984; Su, Wen-King

5124:TR:84 "The Probe: An Addition to Communication Primitives," March 1984; Martin, Alain

5123:TR:84 "The Mossim Simulation Engine Architecture and Design," March 1984; Bill Dally

5122:TR:84 "Submicron Systems Architecture (ARPA Semiannual Technical Report)," March 1984;

5120:TM:84 "A Mathematical Approach to Modeling the Flow," March, 1984; Johnsson, Lennart and Danny Cohen

5118:TR:84 "SMART User's Guide," February, 1984; Ngai, John

5117:DF:84 "Bit Serial Multiplication over Finite Fields," February, 1984; Whiting, Doug

5113:TR:84 "The $Wo_{L}ery$," January, 1984; Mead, Carver A.

5112:TR:83 "Parallel Machines for Computer Graphics," Ph.D. Thesis, January, 1983; Ulner, Michael

5110:DF:83 "Distributed RC Delay Line Model & MOS PLA Timing," December, 1983; Chiang, Chao-Lin

5109:DF:83 "The DSIM Functional Simulation System Preliminary User's Manual," December, 1983; Dally, Bill

5108:DF:83 "A Fast Quaternary Serial Multiplier in CMOS/SOS," December, 1983; Dally, Bill

5107:DF:83 "Stochastic Models for Channel Routing Track Demand," December 1983; Ngai, John

5105:TR:83 "Memory Management in the Programming Language ICL," November

5104:TR:83 "Graph Model and the Embedding of MOS Circuits," MS Thesis, November 1983; Ng, Tak-Kwong

5103:TR:83 "Submicron Systems Architecture (ARPA Semiannual Technical Report)," November 1983;

5102:TR:83 "Experiments with VLSI Ensemble Machines," October 1983;
Seitz, Charles L.

5101:TM:83 "Concurrent Fault Simulation of MOS Digital Circuits," October 1983
Bryant, Randal E.

5100:DF:83 "MOSSIM Simulation Engine Preliminary Architecture," October 1983;
Dally, Bill and Randy Bryant

5099:TM:83 "VLSI and the Foundations of Computation," September 1983;
Mead, Carver

5098:TM:83 "New Techniques for Ray Tracing Procedurally Defined Objects," September 1983;
Kajiya, James T. [$2.00] 13 pages

5097:TR:83 "The Design of a Self-timed Circuit for Distributed Mutual Exclusion," September 1983;
Martin, Alain J.

5096:DF:83 "MOS-VLSI Circuit Simulation Formulations for Concurrent Execution," August 1983;
Mattisson, Sven

5095:DF:83 "A System Programmer's Guide to Cosmic Kernel ," August 1983;
Seitz, Chuck and Athas, Bill

5093:TR:83 "Design of the MOSAIC Element," July 1983;
Lutz, Chris, Steve Rabin, Chuck Seitz and Don Speck 10 pages

5092:TM:83 "Residue Arithmetic & VLSI," July 1983;
Chiang, Chao-Lin & Lennart Johnsson [$2.00] 5 pages

5091:TR:83 "Race Detection in MOS Circuits by Ternary Simulation," June 1983;
Bryant, Randal E. [$2.00] 12 pages

5090:TR:83 "Space-Time Algorithms: Semantics and Methodology,"Ph.D. Thesis, June 1983;
Chen, Marina Chien-mei [5.00] 109 pages

5089:TR:83 "Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circuits," July 1983;
Lin, Tzu-Mu and Carver A. Mead [$10.00] 68 pages

5088:TM:83 "infinite Fair Shuffle," June 1983;
Choo, Young-il

5087:TM:83 "Concurrency Algebra and Petri Nets," June 1983;
Choo, Young-il

5086:TR:83 "A VLSI Combinator Reduction Engine,"MS Thesis May 1983;
Athas, William C., Jr. [$4.00] 40 pages

5085:DF:83 "Solution Set of AM/CS146," May 1983;
Johnsson, Lennart and Peggy Li

5084:TM:83 "The Tree Machine: An Evaluation of Strategies for Reducing Program Loading Time," August 1983;
Li, Pey-yun Peggy, and Lennart Johnsson [$2.00] 26 pages

5083:DF:83 "Role of Parameters-Sticks Representations," May 1983;
Trimberger, Steve

5081:TR:83 "RTsim - A Register Transfer Simulator," April 1983;
Lam, Jimmy [$4.00] 63 pages

5079:TR:83 "Highly Concurrent Algorithms for Solving Linear Systems of Equations," April 1983;
Johnsson, Lennart [$2.00] 79 pages

5078:TR:83 "Submicron Systems Architecture," April 1983;
ARPA Semiannual Technical Report [$5.00] 32 pages

5077:DF:83 "Pooh: A Uniform Representation for Circuits," March 1983;
Whitney, Telle

5076:DF:83 "The Semantics of a Functional Language for VLSI Systems," March 1983;
Chen, Marina

5075:TR:83 "A General Proof Rule for Procedures in Predicate Transformer Semantics," March 1983;
Martin, Alain [$2.00] 20 pages

5074:TR:83 "Robust Sentence Analysis and Habitability," Trawick, David J., Ph.D. Thesis, February 1983

5073:TR:83 "Automated Performance Optimization of Custom Integrated Circuits," Trimberger, Stephen, Ph.D. Thesis, March 1983

5068:TM:83 "A Hierarchical Simulator Based on Formal Semantics," Proc. Third Caltech Conf. on VLSI, p. 207-223, March 1983; Chen, Marina and Carver Mead [$1.00]

5065:TR:82 "Switch Level Model & Simulator for MOS Digital Systems," December 1982; Bryant, Randal E. [$3.00]

5059:TM:82 "A Comparison of MOS PLAs," December 1982; Trimberger, Stephen [$2.00]

5055:TR:82 "FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems," MS Thesis, December 1982;
Ng, Charles H. [$5.00]

5052:TR:82 "Submicron Systems Architecture," submitted to ARPA October 1982; *Semiannual Technical Report* [$4.00]

5049:TR:82 "Distributed Mutual Exclusion Algorithms," submitted for publication, AJM 31, September, 1982; Martin, Alain [$3.00]

5047:TR:82 "The Torus: An Exercise in Constructing a Processing Surface," Proc. 2nd Caltech Conference on VLSI, Caltech, Pasadena CA, January 1981; Martin, Alain [$3.00]

5046:TR:82 "An Axiomatic Definition of Synchronization Primitives," Acta Informatica 16, pp. 219-235 (1981); Martin, Alain [$3.00]

5045:TM:82 "A Distributed Implementation Method for Parallel Programming," Proc. Information Processing '80, S. H. Lavington, (ed.); Martin, Alain [$3.00]

5044:TR:82 "Hierarchical Nets: A Structured Petri Net Approach to Concurrency," September, 1982; Choo, Young-Il [$10.00]

5043:TM:82 "A Formal Derivation of Array Implementations of FFT Algorithms," Proc. USC Workshop on VLSI & Modern Signal Processing, (sponsored by ONR) Nov. 1982, to be published by Prentice-Hall; Johnsson, Lennart and Danny Cohen [$3.00]

5042:TR:82 "Concurrent Algorithms as Space Time Recursion Equations," September, 1982; Chen, Marina and Carver Mead [$4.00]

5040:TR:82 "Concurrent Algorithms for the Conjugate Gradient Method," September, 1982; Johnsson, Lennart [$4.00]

5038:TM:82 "A New Channel Routing Algorithm," September, 1982; Chan, Wan S. [$4.00]

5035:TR:82 "Type Inference in a Declarationless, Object-Oriented Language," MS Thesis, June 1982; Holstege, Eric [$10.00]

5034:TR:82 "Hybrid Processing," Ph.D. Thesis, March 1982; Carroll, Chris [$12.00]

5033:TR:82 "MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual," August 1982; Schuster, Mike and Randal E. Bryant [$4.00]

5030:TM:82 "VLSI Algorithms for Doolittle's, Crout's, and Cholesky's Methods," Proc. ICCC '82, IEEE Int'l Conf. on Circuits & Computers, NY Sept. 1982, pp.372-377, IEEE Catalog No. 82CH1813-5; Johnsson, Lennart [$1.00]

5029:TM:82 "POOH User's Manual," July 1982,; Whitney, Telle [$4.00]

5027:TM:82 "Concurrent Programming," July 1982; Bryant, Randal E. and Jack B. Dennis [$3.00]

5021:TR:82 "Earl: An Integrated Circuit Design Language," MS Thesis, May 1982; Kingsley, Chris [$5.00]

5019:TR:82 "A Computational Array for the QR-Method," Proc. MIT Conference on Advanced Research in VLSI, P. Pennfield, ed., Boston, January 1982, pp.123-129; Johnsson, Lennart [$3.00]

5018:TM:82 "Filtering High Quality Text for Display on Raster Scan Devices," August 1981; Kajiya, Jim and Mike Ullner [$2.00]

5017:TM:82 "Ray Tracing Parametric Patches," May 1982; Kajiya, Jim [$2.00]

5016:TR:82 "Bristle Blocks - Scrutinized and Analyzed," June 1982; McNair, Richard, and Monroe Miller [$4.00]

5015:TR:82 "VLSI Computational Structures Applied to Fingerprint Image Analysis,"; Megdal, Barry [$4.50]

5014:TR:82 "The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture," Ph.D Thesis, April 1982; Lang, Charles R. Jr., [$15.00]

5012:TM:82 "Switch-Level Modeling of MOS Digital Circuits,"; Bryant, Randal [$2.00]

5005:TM:82 "Chip Assembly Tools,"; Trimberger, Steve and Chris Kinglsey [$2.00]

5004:TM:82 "Riot - A Simple Graphical Chip Assembly Tool,"; Trimberger, S. and Jim Rowson [$2.00]

5003:TM:82 "Pipelined Linear Equation Solvers & VLSI ," Proc., Microelectronics 1982, Adelaide, Australia, May 1982, pp.42-47, The Institution of Engineers, Australia, Nat'l Conf. Pub. No. 82/4; Johnsson, Lennart S. [$2.00]

5001:TR:82 "Minimum Propagation Delays in VLSI ," IEEE J. Solid State Circuits, Vol. SC-17, No. 4, August 1982, pp. 773-775; (later version of 4601:TM:81); Mead, Carver, and Martin Rem [$2.00]

5000:TR:82 "A Self-Timed Chip Set for Multiprocessor Communication," MS Thesis, February 1982; Whiting, Douglas [$6.00]

4778:TM:82 "Testing and Structured Design," Proc. International Test Conference., Cherry Hill NJ, August 1982; DeBenedictis, Erik P., and Charles L. Seitz [$2.00]

4777:TR:82 "Techniques for Testing Integrated Circuits,"; DeBenedictis, Erik P.

4724:TR:82 "Concurrent, Asynchronous Garbage Collection Among Cooperating Processors," (superceded by 5014:TR:82); Lang, Charles R. [$5.00]

4716:TM:82 "A Rectangular Area Filling Display System Architecture,"; Whelan, Dan [$4.00]

4710:TM:82 "Earl: An Integrated Circuit Design Language," (Succeeded by 5021:TR:82); Kingsley, Christopher [$5.00]

4684:TR:82 "A Characterization of Deadlock Free Resource Contentions,"; Chen, Marina, Martin Rem, and Ronald Graham [$3.00]

4682:TR:81 "Earl: An Integrated Circuit Design Language," (Succeeded by 5021:TR:82); Kingsley, C. [$3.00]

4675:TR:81 "Switching Dynamics," MS Thesis,; Lewis, Robert K. [$7.00]

4655:TR:81 "Proceedings Second Caltech Conference on Very Large Scale Integration," 19-21 January 1981; Seitz, Charles, ed. [$20.00]

4654:TR:81 "A Versatile Ethernet Interface," MS Thesis; Whelan, Dan [$12.00]

4653:TR:81 "Toward A Theorem Proving Architecture," MS Thesis; Lien, Sheue-Ling [$10.00]

4618:TM:81 "The Tree Machine Operating System,"; Li, Peggy [$5.00]

4601:TM:81 "Minimum Propagation Delays in VLSI," Proc., Second Caltech Conf. on VLSI, January 1981; Mead, Carver A. and Martin Rem [$3.00]

4600:TM:81 "A Notation for Designing Restoring Logic Circuitry," Proc., Second Caltech Conf. on VLSI, January 1981; Rem, Martin, and Carver A. Mead, (revised from 4529:TM:81) [$3.00]

4530:TR:81 "Silicon Compilation," Ph.D. Thesis; Johannsen, Dave [$18.00]

4521:TR:81 "Lambda Logic," MS Thesis; Rudin, Leonid [$8.00]

4517:TR:81 "The Serial Log Machine," MS Thesis; Li, Peggy [$7.00]

4407:TM:82 "An Experimental Composition Tool,"; Mosteller, Richard C. [$3.00]

4379:TR:81 "LAP User's Manual,"; Lang, D. (Rev. by C. Kohle and S. Trimberger 9/81) [$2.00]

4336:TR:81 "A Structured Design Methodology and Associated Software Tools,"; Trimberger, S., J. Rowson, D. Lang, and J.P. Gray [$4.00]

4334:TR:81 "An Inexpensive Multibus Color Frame Buffer,"; Whelan, Dan and R. Eskanazi [$1.00]

4332:TR:81 "RLAP, Version 1.0, A Chip Assembly Tool,"; Mosteller, R. [$3.00]

4320:TR:81 "A Hierarchical Design Rule Checker," MS Thesis; Whitney, Telle [$7.00]

4317:TR:81 "REST - A Leaf Cell Design System," MS Thesis; Mosteller, Richard C. [$10.00]

4298:TR:81 "From Geometry to Logic," MS Thesis,; Lin, Tzu-mu [$5.00]

4287:TR:81 "Computational Arrays for Band Matrix Equations,"; Johnsson, L. [$2.00]

4281:TR:81 "Combining Graphics and a Layout Language in a Single Interactive System (2nd Revision),"; Trimberger, S. [$2.00]

4270:TR:81 "FIFI Test System Preliminary User's Manual," (later version contained in 4777:TR:82); DeBenedictis, Erik [$3.00]

4204:TR:78 "A 16-Bit LSI Digital Multiplier," EE Thesis; Masumoto, R. T. [$8.00]

4191:TR:81 "Towards A Formal Treatment of VLSI Arrays," Proc., Second Caltech Conference on VLSI, Pasadena, CA, January 1981; Johnsson, Lennart S., Uri Weiser, D. Cohen, and Alan L. Davis [$4.00]

4168:TR:81 "Computational Arrays for the Discrete Fourier Transform," Proc., 22nd Computer Science Int'l. Conference, CompCon 81, San Francisco, February 1981, pp. 236-244, IEEE Catalog No. 81CH1626-1; Johnsson, Lennart S. and D. Cohen [$3.00]

4088:TR:80 "The Representation of Communication and Concurrency,"; Milne, George [$8.00]

4087:TR:80 "Gaussian Elimination of Sparse Matrices and Concurrency - A Complexity Analysis," (Succeeds 4087:DF:80); Johnsson, Lennart [$3.00]

4061:TR:80 "A Preliminary Report on the Caltech ARPA Tester Project," (later version in 4777:TR:82); DeBenedictis, Erik [$4.00]

4029:TR:80 "Structure, Placement and Modeling," MS Thesis; Segal, R. [$8.00]

4025:TM:80 "Sticks Standard Software Package,"; Segal, R. and Steve Trimberger [$2.00]

4024:TM:80 "SSP Basic Software Package (Revised),"; [$5.00]

4022:TM:80 "Comprehensive CIF Test Set (Revised),"; Trimberger, S. [$5.00]

3901:TM:78 "Hierarchical Design for VLSI,"; Rowson, J. [$3.00]

3882:TM:80 "A Chip Assembler,"; Tarolli, Gary [$2.00]

3880:TM:80 "The Proposed Sticks Standard,"; Trimberger, S. [$5.00]

3857:TM:80 "VLSI Architecture & Design," Proc., National Electronics Conference, Chicago, Oct., 1980, Vol. 34, pp. 254-259; Johnsson, Lennart [$1.00]

3805:TR:80 "SSP Annual Report,"; Silicon Structures Project [$3.00]

3762:TR:80 "A Software Design System," Ph.D. Thesis; Hess, Gideon [$8.00]

3761:TR:80 "A Fault Tolerant Integrated Circuit Memory," Ph.D. Thesis; Barton, Tony [$7.00]

3760:TR:80 "The Tree Machine: A Highly Concurrent Computing Environment," Ph.D. Thesis,; Browning, Sally [$10.00]

3759:TR:80 "The Homogeneous Machine," Ph.D. Thesis; Locanthi, Bart [$7.00]

3710:TR:80 "Understanding Hierarchical Design," Ph.D. Thesis; Rowson, James [$9.00]

3642:TM:80 "Modeling and Verification in Structured Integrated Circuit Design," Ph.D. Thesis; Buchanan, Irene [$10.00]

3487:TM:80 "The Proposed Sticks Standard,"; Trimberger, Steve [$2.00]

3364:TM:79 "Stack Data Engine," December 1979; Efland, G. and R. C. Mosteller [$5.00]

3357:TM:79 "CIF2OP Instruction Manual,"; Tarolli, Gary and Dick Lang [$3.00]

3356:TR:79 "LAP User's Manual,"; Lang, Dick [$3.00]

3353:TM:79 "FORTRAN Debugging Aids,"; Trimberger, Steve [$3.00]

3352:TM:80 "A Comprehensive CIF Test Set," December 1979; Trimberger, Steve [$2.00]

2883:TR:79 "A Pascal Machine Architecture Implanted in Bristle Blocks, a Prototype Silicon Compiler," MS Thesis; Seiler, Larry [$10.00]

2870:TM:79 "A Wire Oriented Mask Geometry Editor,"; Trimberger, Steve [$5.00]

2686:TR:80 "The Caltech Intermediate Form for LSI Layout Description,"; Sproull, Robert and Richard Lyon, revised by S. Trimberger [$2.00]

2276:TM:78 "A Language Processor and a Sample Language,"; Ayres, Ron [$12.00]

1584:TM:78 "Cost and Performance of the VLSI,"; Mead, Carver A. [$3.00]

1438:TM:78 "Polygon Package,"; Sutherland, Ivan E. [$3.00]

APPENDIX A

DRAFT
To be published J. VLSI & CS, vol 1, no 3, Computer Science Press, 1984

Experiments with VLSI Ensemble Machines*

Charles L Seitz†

Abstract – **Our ongoing research at Caltech in VLSI architecture and design is entering a phase in which we are doing experiments with the design, programming, and applications of working concurrent systems of useful size and performance. The three systems to be discussed, *cosmic cube*, *mosaic*, and *super mesh*, are all structured as ensembles of identical elements that operate concurrently and communicate by message passing. These systems differ in many significant respects: element size, communication plan, MIMD vs SIMD, and consequently, in programming style and applications. This paper presents the common principles and issues that have guided these designs, reports preliminary results, and discusses plans for research and system building experiments that we hope and expect will yield significant results over the next several years.**

1. INTRODUCTION

Considerations at many levels, from VLSI design to applications, have led the research in VLSI architectures at Caltech to investigations of a family of programmable machines that are ensembles [Seitz82] of identical, concurrently operating, small computers. The term "ensemble" is meant to suggest that the computers perform as a group, in close cooperation or harmony, similar to a musical ensemble. The computation to be performed is orchestrated by the programmer. Improvisation is possible, and is interesting to think about, but will not be discussed here. These machines do not run ordinary sequential programs.

In order that these machines be strictly scalable in VLSI implementations, the individual computers are small and communicate only by queued message passing. There is no storage or other system resource that is global or shared except for the communications. The individual computers are referred to as "nodes", as in a computer network. Messages may be sent between the nodes over channels in a regular communication plan such as a tree, mesh, shuffle, or hypercube.

In the MIMD machines, messages are queued and may be routed through intermediate nodes on the way to nodes that are not neighbors in the communication graph. Thus a multiple process message passing computational model, a process or object oriented programming style, and simple extensions of common sequential programming notations, are natural for these machines. This

*The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

†Computer Science 256-80, California Institute of Technology, Pasadena CA 91125.

comparatively unrestrictive model has proved to be quite satisfactory for the numerically demanding physical simulation and modeling applications that have been studied and implemented to date.

It is not strictly necessary to the structural or computational model that these MIMD machines be homogeneous, that is, composed of only one type of node, or that the communication plan be regular. Such machines are simply easier to design, build, program, and maintain than heterogeneous and/or irregularly connected ensembles.

The SIMD machine that we have designed and simulated is quite different from the MIMD machines. It is necessarily homogeneous. Communication between nodes is synchronized by the broadcast instruction stream, and the communication graph is restricted for physical reasons to a mesh. Its applications are highly regular computations that can be described by sequential programs that operate on array structures.

Our early investigations of algorithms and programming disciplines for machines based on these principles [*Browning80, Locanthi80, Browning&Seitz81, Lang82*], including extensive simulations [Lang82, Li82], left many questions open that could not be addressed by simulation. The dangerous itch to give our ideas a realistic test, and design studies that indicated that such machines were quite feasible to build in a university and offered extraordinary cost/performance characteristics, led us to initiate three system building experiments in the 1980–1982 period. The following paragraphs briefly summarize the structure and status of these machines.

Cosmic cube, an experimental MIMD homogeneous machine with 64 nodes connected in a Boolean 6-cube, is in operation and regular use. The complexity of the present cosmic node, which includes 136K bytes of storage, 16-bit instruction processor, floating point coprocessor, and 6 communication channels, is representative of a node that could be integrated onto a single chip with about 1 micron feature size technology. Thus we think of the cosmic cube as a system that could be built with single chip nodes in about 5 years from now. In anticipation of this advanced process technology, we have built a "hardware simulation" of the nodes from commercially available integrated circuit components in order to experiment with the applications, algorithms, programming, and engineering of such systems. This 64 node machine has sufficient performance, about 10 times a VAX11/780 or 1/10th of a Cray-1, on problems for which it is suited that it has attracted many Caltech scientists to become users, which is indeed part of our experiment. Also, the nodes have sufficient storage for interesting experiments with distributed operating systems. More advanced versions of this style of machine are now being designed.

It was a premise of the *mosaic* experiment that the node would be a single chip [Lutz84] with as much capability and performance as we could achieve with acceptable yield in standard MOSIS [*Cohen81, Lewicki84*] fabrication. These nodes are thus much smaller, but are otherwise identical in structure to the cosmic node. Each node includes only 4K bytes of storage with a 16-bit processor and 4 communication channels. Mosaic nodes can be connected in a variety of fixed degree (≤ 4) communication plans, including the tree, mesh, chordal ring, shuffle, and cube connected cycle. The single-chip version of this node has not yet been fabricated, pending investigations of its yield from sample yields of its principal parts. However, small mosaic systems using a version of the mosaic node with processor and communication channels on a MOSIS chip together with 8K bytes of fast commercial storage chips have been assembled and run test programs. We expect to use the single chip node to assemble a 1024 node mosaic shuffle or mesh during 1985. This 1024 node system will be capable of executing several billion instructions per second, or about 50 million (32-bit mantissa) floating point operations per second, which is quite remarkable for a system of only 1024 chips

packaged into a cubic foot or so. Even with this meager storage capacity, the application span of mosaic includes many highly concurrent problems in signal and image processing, and finite element and matrix computations.

A fine grain SIMD array machine dubbed *super mesh* has been designed and simulated, and its very small repeated node is being laid out as a custom chip. Multiple nodes per chip are readily achieved. Although this "array" machine is aimed at very high floating point performance and efficiency, it should not be confused with a "peripheral array processor." Rather, its array structure makes it similar to Illiac IV [Bouknight72], STARAN [Batcher74], the ICL DAP [DAP], or to a systolic [Kung80] or computational array in which (unlike the CMU programmable systolic chip [Kung84]) the nodes operate from a single, or at most several, instruction streams. This project is aimed at an extreme point of cost/performance versus generality, and is more an experiment in VLSI design, synchronization over large volumes, and optimal arithmetic algorithms, than in architecture and programming. *Although we can report only simulation results for this machine, we expect it* to exhibit extraordinary cost/performance characteristics, but for a limited class of highly regular computational problems.

One must state immediately the precaution that even the larger grain versions of these experimental machines are not claimed to be efficient for all types of computations performed on conventional mainframes. It is not yet known, and is certainly one of the objectives of our research to discover, how far we can extend these machines toward irregular applications.

2. PRINCIPLES and DESIGN ISSUES

These three system building experiments have a certain consistency in principles of design, operation, and programming, outlined in the subsections below.

2.1 Cost/performance of VLSI Architectures

It is the cube law scaling of the switching energy with feature size in constant electric field scaling of MOS technologies [Dennard74, Mead&Conway80] that is the payoff, promise, and driving force for the advances in these technologies. Scaling feature size and voltage in MOS systems reduces the power and area per device quadratically with the scaling factor, and thus reduces the cost by approximately this factor. The power per unit area is held constant in this scaling. The transit time and circuit delays are reduced linearly with the scaling factor. The product of the power per device and device delay, which is equivalent to the energy for each switching event, thus scales by a cube law. The switching energy or power delay product is a fundamental figure of merit of a technology that has a direct relation to the cost/performance of systems implemented in that technology.

We have tried to apply the very stringent requirement on a "VLSI architecture" that the cost/performance of a system scale as closely as possible with the switching energy. This scaling is difficult to achieve in practice in large, tightly-coupled systems due to the dominant contribution of communication in area, power, and delay. At the risk of trying to demonstrate the counterpart of the famous proof that bumblebees cannot fly, analyses of scaling large, tightly-coupled systems, including the cost in area and energy of long distance communication on and between chips, the communication delay in drivers, and the delay in the diffusive propagation of signals in scaled wires *[Seitz79, Mead&Rem 4601:TR:81, Bilardi82], indicate that there are diminishing returns in*

performance for area. In these cases the scaling of transit time in the MOS switches and small MOS circuits is not reflected in system performance because the delay of the interconnect does not scale.

At the same time, we insist that the architectures be open-ended in performance, but see few opportunities for performance improvements in single process systems. The instruction execution rates of computers ranging from single chips to supercomputers are surprising tightly bunched, predominantly between 1 and 100 million instructions per second (Mips), even though the range of costs is spread across more than 5 orders of magnitude. (Granted, a Cray-1 instruction is not only executed 100 times faster than an 8-bit micro's, but also on much longer words, but the difference in word size relates to the parallelism rather than to instruction *rate*.) Other evidence for this performance limit, or diminishing returns in performance for cost, is that the rate of improvement in execution bandwidth of processors on the high-speed end of the spectrum has been decreasing [Buzbee83]. In documenting this trend, Buzbee has conjectured from a mathematical fit to the history of computer performance that these high-speed single process machine designs are within about an order of magnitude of an asymptote.

For ensemble machines these requirements of (1) cost/performance scaling with technology advances, and (2) open-ended performance, imply that systems be scalable to very large numbers of nodes without sacrificing cost/performance. These requirements are satisfied directly by the limited size and loose coupling of the nodes. However, this family of machines is clearly more specialized than the usual sequential computer, but to a degree that can only be answered by doing the experiment.

2.2 Performance Leverage in Concurrency

In order to see what kind of performance can be achieved in cost effective designs with systems based on the high complexity but modest performance MOS technology, one inevitably looks to systems that support multiple concurrent processes. Major advances in performance, say factors of 1000 over today's fastest computers, evidently require that the degree of concurrency be very large indeed. Such goals require more than the evolution of the single process computer to the multiprocessor, but to machine organizations based on very different algorithms, principles of operation, and programming.

Today's fastest computers, and even many microprocessors, exploit concurrencies that can be discovered in the process of interpreting an instruction stream. Techniques such as instruction prefetch, overlap or pipelining of arithmetic operations, and caching to exploit locality in storage references, allow typically and in aggregate a 10-fold concurrency. These machine designs have been tuned to exploit the typical concurrency that can be found in programs [Kuck74] by optimization, vectorization, and decomposition of the sequential program to fit concurrent processing elements.

Ensemble machines can achieve strikingly higher degrees of concurrency from concurrent formulations of the whole computation. We do not perform automatic decomposition and vectorization of existing sequential programs, except to the extent that our compilers and processors can exploit the concurrencies in the innermost iterations and expression evaluations in the code that defines individual processes. Rather, we want to turn the size of a large computation against itself. Many massive computations — eg matrix computations, differential equation solution, many-body problems, Fourier transforms, simulation of multiple degree of freedom systems, image and signal processing — have natural concurrent formulations whose degree of concurrency is large and grows with the problem size.

Since we do not know how to automate the discovery of efficient concurrent formulations for

such problems, we leave to the programmer the task of inventing and expressing a concurrent and spatially distributed algorithm for a whole computational problem. It will be interesting to learn how far we can go with this approach, and whether the programming task can be simplified by starting, for example, from a mathematical [Johnsson81] rather than procedural specification of the problem.

2.3 Message passing

The model of computation, and the target for the problem formulation, for the ensemble machines discussed here is a collection of sequential processes that communicate by message passing. A static process structure, or snapshot of a dynamic process structure, can be represented as a graph of process vertices connected by arcs (directed edges) that represent reference. (When process A possesses reference to process B, a message from A to B is possible.) The hardware communication structure of these machines can be represented similarly, and the mapping of a computation onto an ensemble machine is an embedding of the computation graph onto the machine graph. The embedding reveals both the locality of communication achieved and load balancing properties of any mapping.

In SIMD array machines message passing is a highly synchronized parallel exchange of data. However complex the computation in a node may be, these machines readily accommodate only problems with quite regular formulations that fit their hard-wired communication structure.

Unlike SIMD array machines, which are effectively limited to problems with these regular formulations, the MIMD message passing ensembles can deal also with computations with highly irregular, and even dynamically changing, communication topologies. Here the message passing model seen by the programmer may be substantially abstracted from the hardware structure, in two ways. First, messages are routed in the communication subsystem, so that formulations need not fit any particular hard-wired communication structure. Second, synchronization between message sending and receiving is potentially quite loose, due to queueing of messages in the communication subsystem.

This message passing approach organizes multiple computers into a structure similar to a computer network. The queue is a better fit to VLSI engines than the switch required for shared storage multiprocessors, for many of the same reasons that store and forward packet communication is used in digital communication networks instead of circuit switching [Kleinrock74]. As an engineering technique, queued message passing has the advantages over circuit switching (1) that it scales well with system size, because the cycle time of the node need not increase with the size of the system, and (2) that it allows messages to be queued and routed in transit in a way that increases throughput even though it increases latency.

Some proposed concurrent systems produce the appearance of sharing to the programmer, while the underlying mechanisms work by message passing. In our experiments we have made the choice of treating message passing as an explicit primitive, with the result that the communication is expressed directly in a problem formulation or program.

2.4 Applications and Performance Modeling

Caltech scientists in high energy physics, astrophysics, quantum chemistry, fluid mechanics, structural mechanics, seismology, and computer science, are developing concurrent application programs to run on these ensemble machines. The first research paper on scientific results, a quantum field theory computation on a cosmic cube prototype [Brooks83], has already been published, and

other applications are developing rapidly. Several of us in computer science are involved in this research both as the system builders and also through interests in concurrent computation and applications to VLSI analysis tools.

The general characteristic of all of these computations is that the sequential part is very small. Following a model by Ware [Ware73], "speedup", S, can be defined as:

$$S = \frac{\text{time on 1 node}}{\text{time on N nodes}}.$$

Ware's formula for speedup:

$$S = \frac{1}{(1-\alpha)+\alpha/N},$$

where "α" is the fraction of the computation that can be done concurrently, follows directly from the definition; eg, for unit time on one node, with N nodes the concurrent part of the computation requires time α/N, but the part that resists being done concurrently still requires time $(1-\alpha)$. This formula reveals the same limitation represented by "Amdahl's argument" [Amdahl67], that speeding up a concurrent computation, even in the limit where N becomes arbitrarily large, is limited to $1/(1-\alpha)$, the reciprocal of the fraction of the computation that must be done sequentially.

We are usually less interested in the asymptotic speedup than in the region in which speedup is almost equal to N, since here an ensemble machine is operating with high efficiency, that is, with very few of the nodes being idle. Such efficiency is achieved with concurrent formulations in which the sequential fraction $(1-\alpha)$ is sufficiently small, say .0001, that its reciprocal, 10000, is a number substantially larger than the number of nodes applied to the problem. In fact, the sequential fraction of a computation turns out not to be a serious limitation for a large class of important problems. Often $\alpha = 1$, except for the program loading and data I/O phases of a computation.

Once this condition of $\alpha \approx 1$ is satisfied, the speedup obtained by using N nodes concurrently is limited by considerations of (1) idle time due to poor load balancing, (2) waiting time due to the large communication latencies that we permit in the physical design of these machines, and (3) processor time dedicated to processing and forwarding messages. This third consideration can be effectively eliminated by architectural improvements in the nodes, but the other considerations are fundamental to the formulation of a computation and to the assumptions we make about the physical design of these machines.

Analyses of problem formulations or programs provide simple and exact predictions for the performance of highly regular physics computations [Fox&Otto84], in which load balancing is easily assured. Analyses of computations that are less regular and more sensitive to latency necessarily use much more complex models of the computation in order to account for variations in process running times, imperfect embeddings and load balancing, and latency. (However, these analyses still show $S \approx N$ for small N and an asymptotic S for large N.) For example, for MOS-VLSI circuit simulation [Mattisson83], the model evaluation (linearization) phase of the computation, which is typically over 80% of the running time on a sequential computer, shows $\alpha = 1$ and a N-fold speedup on a concurrent machine. However, solving the sparse matrix equations and communicating the intermediate results between parts of the computation in a circuit of arbitrary topology presents a very complex analysis problem. Amongst the many unknowns in experimenting with circuit simulation, as a paradigm of less regular computations that can be performed on ensembles, are the interaction between communication cost and load balancing in the mapping of processes to nodes.

Processes here correspond with rows in a sparse but "clumped" admittance matrix. Although the "clumping" can be exploited in this mapping to localize communication, it may also concentrate many of the longer iterations occurring during a signal transient into a single node, thus creating a "dynamic" load imbalance in the computation.

2.5 Grain Size

The attraction of ensembles with smaller nodes, "fine grain" systems, is their cost/performance characteristics. Since the instruction rate of a smaller node can be expected to be comparable to, or with shorter signal paths, even faster than that of the larger node, the fine grain node provides similar performance at less cost. At the same system cost, the smaller node is the basis for an ensemble with more nodes, more performance, similar total storage, and less storage per node. When this tradeoff is applied to comparisons of "fine grain" machines, such as mosaic or super mesh, and "medium grain" machines such as the cosmic cube, one expects [Seitz82] that the fine grain machines will have a smaller application span, for two reasons.

First, if the cost/performance advantage of small nodes is to be reflected in the cost/performance of the computation, one cannot use so many nodes that the speedup is in its asymptotic region. Machines with many nodes are efficient only on computations of commensurate size and concurrency. Properly, this problem-dependent limit on the number of nodes that can usefully be kept busy applies as well to coarse grain machines with many nodes, or to comparisons between single process computers and ensembles, and relates to grain size only for comparisons of systems of constant cost.

Second, the small storage of a fine grain node limits the size of a process, either the program complexity (MIMD) or the data (MIMD and SIMD), or both. Progressively finer grain machines violate more and more seriously the old rule of thumb that instruction processors require about a Mbyte of storage per Mips execution rate. A partial reason that we get away with less data storage than one might expect from the node instruction rate, even on extremely protracted computations, is that the data being processed is not all stored in the node, but is also communicated. The working set can grow with the machine size. However, the size and complexity of the concurrent processes—the grain size of the formulation—is generally bounded by the node size.

Optimal cost/performance for a given computation is achieved with a machine with the exact number of nodes required, each exactly large and fast enough for the node process, a clearly unrealistic expectation for other than very regular and specialized problems. Thus even with a family of machines of different grain size, which is what we seek as a suitable experimental environment, formulations that are of substantially finer grain than that of the target ensemble are the interesting case for attempting to extend the generality of this family of machines. Techniques for mapping processes onto the ensemble [Martin81], many processes per node, or many nodes per process, either statically or in execution, assume a central role in extending highly concurrent computations beyond regular and specialized problems.

3. COSMIC CUBE EXPERIMENT

3.1 Chronology

The cosmic cube design is based in largest part on extensive program modeling and simulations by Charles R (Dick) Lang [Lang 5014:TR:82] carried out between 1980 and 1982. In particular, it was from this work that the communication plan of a Boolean n-cube, the bit rates of the communication

channels between nodes, and the organization of the operating system primitives were chosen. In earlier work at Caltech [Locanthi 3759:TR:80] and from interactions with Carl Hewitt [Hewitt80] at MIT, including an August 1981 meeting organized by Hewitt, we recognized the similarity of our ideas to those presented in a paper by Herbert Sullivan [Sullivan77], and so refer to cosmic cube by the term coined by Sullivan, a *homogeneous machine*, a machine "of uniform structure."

The logical design of the cosmic node was done in the fall of 1981 by Erik DeBenedictis. Most of the choices made in this design are fairly easy to explain. First of all, a Boolean n-cube communication plan was used because this network was shown by simulation to provide very good routing properties. It also contains all meshes of lower dimension, which is useful for regular mesh-connected problems, and the connections required to map FFT algorithms directly. The Boolean n-cube can be viewed recursively; that is, the n-cube that is used to connect $2^n = N$ nodes is assembled from two $(n-1)$-cubes, with corresponding nodes connected by an additional channel. This property simplifies the packaging of machines of varying size. It also explains some of the excellent message flow performance properties of the Boolean n-cube on irregular problems. The number of channels connecting the pairs of subcubes is proportional to the number of nodes, and hence on average to the amount of message traffic they can generate.

The Boolean n-cube, which is often referred to in the parallel processing literature as a direct binary n-cube, is a logarithmic network, like the shuffle [Lawrie75], or various indirect n-cubes such as the banyan [Goke73] or Omega [Lawrie75] network, or the "flip" network used in STARAN [Batcher76], in that the longest path in a machine of N nodes is $\log_2(N) = n$. The shuffle, however, does not contain submeshes, and must be rewired when expanded. The Boolean n-cube is isomorphic to indirect n-cube networks of $(N/2)n$ 2x2 switches that form an NxN switch in n layers, except that the N processing nodes connect to the N paths and route messages at each of the n layers of the indirect network. Of course the n-cube graph is not of fixed degree. The present cosmic cube nodes, with 6 communication channels, cannot be assembled into an n-cube larger than 64 nodes.

With this rich connection scheme, simulation showed that we could use channels that are fairly slow (about 2 Mbit/sec) compared with the instruction rate. The communication latency is, in fact, deliberately large to make this node more nearly a hardware simulation of the situation anticipated for a single chip node. The processor overhead in dealing with each 64 bit packet is comparable to its latency. The communication channels are asynchronous, full duplex, and include queue storage for a 64 bit "hardware packet" both in the sender and receiver in each direction, a basic minimum necessary to decouple the sending and receiving processes.

The Intel 8086 was chosen as the instruction processor because, at the time, it was the only single chip instruction processor available with a floating point coprocessor, the Intel 8087. This floating point performance was necessary for applications that our colleagues in high energy physics at Caltech, under the direction of physics professor Geoffrey Fox, wished to attempt. The storage size of 128K bytes was the subject of a great deal of internal discussion of "balance" in the design. It was argued that the cost incurred in doubling the storage size would better be spent on more nodes. In fact, this choice is clearly very dependent on target applications and programming style. The dynamic RAM includes parity checking but not error correction. Each node also includes 8K bytes of read-only storage for initialization, bootstrap loader, dynamic RAM refresh, and diagnostic testing programs.

A prototype 4 node (2-cube) system on wirewrap boards was assembled and tested in the spring of 1982, and this system has been running concurrent programs and has been in use for software

development since July 1982. The homogeneous structure of these machines is exploited to accelerate the software development by use of a small hardware prototype that is essentially similar to scaled up machines. This same tactic is being used in the mosaic project.

With the design thus proved, we had printed circuit boards designed, and went through the other packaging logistics of assembling a machine of useful size. The present cosmic cube grew from an 8 node to a 64 node machine over the summer 1983, and this 6-cube machine has been in regular operation since October 1983. The task of building hardware to provide more cycles for the user group has been passed to a group at Caltech JPL, with the intention of building approximately 200 more nodes of a program compatible derivative design over the next year. The "mark II" cosmic nodes have 256K bytes of storage and 9 communication channels, and will be configured as a 7-cube and two 5-cubes.

Figure 1 is a photograph of the 6-cube in operation. Larger machines would have nodes arrayed in 2 or 3 dimensions, but for such a small machine, and large ratio of PCB width to spacing, a one-dimensional projection of the 6-cube is satisfactory. The separate units on the shelf above the long 6-cube box are (left to right) the power supply and two "intermediate hosts" (IHs). These dedicated IH machines run network and operating system software, and in some cases are used to support small auxiliary computations in support of the computation running in the cube. The volume of the system is 6 cubic feet, and power consumption is 700 watts. We also operate a 3-cube machine in support of software development, since the 6-cube is not readily shared.

8.2 Early Application Programs and Benchmarks

Programs written by physicists Eugene Brooks and Steve Otto have allowed direct benchmarking of various size cosmic cubes against a VAX11/780. These programs consistently place a 64 node cosmic cube at about 10 times faster than the VAX11/780 on this restricted class of problems. These programs execute somewhat over 3 million 32-bit floating point operations per second. It is estimated that the cosmic cube is about 1/10th as fast as a Cray-1 for these same problems, but would be less than 1/20th a Cray-1 on problems requiring 64-bit floating point computations. The instruction rate and storage bandwidth for computations that do not use floating point compare still more favorably with conventional machines. Thus we believe that for these specialized problems the cosmic cube, which would have about the same leveled out manufacturing cost as a VAX11/780 processor with the same 8 Mbytes of primary storage, achieves a cost/performance advantage over conventional mainframes of about 10.

As an example of its applications at Caltech, a lattice computation programmed by Steve Otto, about 63K bytes of program and 64K bytes of data per node, has now run for more than 2,000 hours on the 6-cube. This program is a Monte Carlo simulation on a 12x12x12x16 lattice, an investigation of the predictions of quantum chromodynamics, a theory that explains the substructure of elementary particles such as protons in terms of quarks and the glue field that holds them bound. After it had run for about 400 hours, this program surpassed the results produced by the most extensive prior computations on this problem, produced in 40 hours on a Cray-1. Otto has shown for the first time in a single computation both the short range Coulombic force and constant long range force [Fox&Otto84]. The communication overhead in this computation, the fraction of the time the nodes spend in the send and receive routines, is only 2.5%.

Amongst the most interesting programs currently in development is a MOS-VLSI circuit simulator [Mattisson 5096:DF:83] that promises very good performance, and is a vehicle for developing tech-

niques for approaching less regular computations on ensemble machines. This program uses a nodal admittance formulation for the electrical network. The admittance matrix is sparse, and "clumped", but because electrical networks have arbitrary topology, does not have the crystalline regularity of the physics computations. The problem is mapped onto the cube by partitioning the admittance matrix in rows, with numerous row processes per node. The linear equation solution phase of the computation, a Jacobi iteration, involves considerable communication, but the linearization (called model evaluation in circuit simulators) that requires about 80%of the execution time on sequential computers is completely uncoupled. Integration and output in computing transient solutions are very small components of the whole computation.

8.3 Programming Environments

Programs for the cosmic cube are developed on conventional computers, written, compiled, simulated, instrumented, and debugged, and then downloaded into the cube through a connection that is managed by the intermediate host. There are 3 layers of software environments in the system:

(1) The lowest level is what we will call the *machine intrinsic* environment. This environment includes the instruction set of the node processor, its I/O communication with channels, and a small initialization and bootstrap loader program stored together with some diagnostic programs in read-only storage in each processor [Newton&Athas 5070:DF:83]. This level is important principally for machine initialization, loading, and diagnostics. It is a relatively more interesting system than one might at first imagine, since a part of the initialization involves each of the identical nodes discovering by sending messages its position in whatever size cube was specified in the startup packet. This initialization involves $n!$ messages that also check the function of all of the communication channels to be used.

(2) The second level environment, referred to as *crystalline*, is characterized by programs written in notations such as C, in which there is a single process per node and in which messages are sent by direct I/O operations to a specified channel. This level has proved to be satisfactory for producing efficient application programs for computations that are so regular that they do not require message routing. The programmer uses a pair of system utilities, RBOX and WBOX, to accomplish message sending and receiving. These routines provide the necessary synchronization, and are also instrumented to collect statistics and to provide error checking on the channels. Although it is perfectly possible for the processes in the nodes to differ, in programs such as Otto's lattice computation they do not. Processes in different nodes take different execution paths depending on the data.

(3) The third level system supports a limited *multiple process* environment with a small operating system called "cosmic kernel" [Seitz&Athas 5095:DF:83] running in each node. Message routing and queueing is accomplished by the cosmic kernel, and messages are sent by reference to a process ID in which is embedded the physical node and process name within that node. This system is limited in the sense that it does not support relocation of processes between nodes during execution. This kernel includes process spawning and destruction, scheduling, storage allocation, and (host) I/O communication. In this environment the definition and execution of the multiple process formulation of the problem is independent of its mapping onto physical nodes; that mapping influences only the efficiency. This environment, when it becomes fully developed, is that intended to be seen by users for both regular and quite irregular application programs.

8.4 Future Homogeneous Machines

Although the present 64 node machine has adequate performance for programming experiments and for accomplishing many useful computations, we are looking ahead to much more powerful systems. Each node for the present cosmic cube requires about 140 $M\lambda^2$ in aggregate complexity across 78 chips, most of which are not high-complexity parts. This node could be expected to scale to a single chip within about the next 5 years. A technology with 1 micron feature size ($\lambda = 0.5$ micron) on chips 6 mm on a side would provide this 140 $M\lambda^2$ complexity. Such a chip is a fairly small increment in complexity over a million bit storage chip. If storage chips and logic continue to be made with different fabrication processes, a single node might better be packaged as two or more chips in a hybrid package. Based on the benchmarks of the cosmic cube, and the expectation that single chip or hybrid nodes would exhibit higher performance, a system of 1024 such chips or hybrids would provide for many computationally demanding problems performance well beyond today's fastest computers. But such thinking is for the long term.

In the short term, as long as we are experimenting with this class of machines by using largely off the shelf parts, most obvious opportunities for improvement (deficiencies) in the cosmic cube design relate to (1) the low level of integration in the communication sections, and (2) context switching between communication and computation.

In order to address the first deficiency, a custom communication chip called the "fifo buffered tranceiver" was designed and tested by Charles Ng [Ng 5055:TR:82] for use in a future homogeneous machine. It provides fifo buffering of 4 256-bit packets each direction in each channel, compared with 2 64-bit packets each direction in cosmic cube, and does full duplex communication using only 2 wires compared with 12 in cosmic cube. The layout is small enough to allow 4 channels per chip.

The second deficiency led to the idea [Athas82] that future descendants of the cosmic cube should use two processors per node, one to manage communication and the other for user processes. This natural partitioning of the tasks of a node not only reduces the problems of context switching in a single processor node, it also allows the operating system kernel to run almost entirely in the communication processor, a considerable advantage since many of the operating system functions can be accomplished concurrently with user processes.

There are three limits to scaling a cosmic cube style of machine, one associated with its connection plan, one associated with reliability, and one associated with the size of the node element.

Boolean n-cubes of progressively higher dimension will be required for larger numbers of nodes. The number of communication channels per node exhibits a relatively benign growth as the $\log_2(N)$. If the node must be physically larger to accommodate more pins, normalized to the number of wires one may place per linear dimension, the structure is always locally wirable, even in two physical dimensions. However, the length of the channels for a machine wired in D physical dimensions doubles each D abstract dimensions, with the result for D limited to two or three (for physical cubes) that the total volume of wire increases more rapidly than the volume simply required to accommodate the nodes. With present wiring technology and in 2 physical dimensions (printed circuit boards and "motherboard" backplanes), it is easy to wire a 14-cube (16K nodes), difficult to wire a 16-cube (64K nodes), and impractical to wire an 18-cube (256K nodes). Should machines larger than 64K nodes ever be built, they would require an advanced interconnect technology or (more likely) a different connection plan.

Reliability limitations appear to be equally distant, at least if soft error control is included. The mean time between failure (MTBF) observed for the cosmic cube node, exclusive of soft errors in its

dynamic RAM primary storage, is well in excess of 100,000 hours. This node MTBF would result in a system MTBF of about a week for a 1K node machine, or an MTBF of an hour or so for a 64K node machine. Even an hour MTBF might be tolerable on a machine whose performance would be about 100 times a Cray-1. Machines of this size are not likely to be built from today's technology, so the reliability situation for future machines is much brighter. The homogeneity of these systems makes faults very easy to diagnose and repair. While it is possible to imagine approaches to fault-tolerant operation, our examinations of this problem suggest that it is not as trivial as it may first appear.

One other scaling consideration is whether the node complexity might reach a point of diminishing returns for performance if the node complexity is scaled up, or equivalently, with a node similar in complexity to the cosmic node in a scaled microcircuit technology. Why shouldn't a node have a Mbyte or more of primary storage? Nodes of this size might well be appropriate for many problems, but since storage can be expected to dominate the silicon area of a fully integrated node, one would expect the cost to increase almost in proportion to the storage size, but with no large change in the instruction rate. Thus one is led to the uncomfortable conclusion that the evolution of systems in the direction of larger nodes is a losing proposition on the basis of cost/performance. Thus we are led to the mosaic class of machines, an experiment with a much smaller node element.

4. MOSAIC EXPERIMENT

Ensembles of single chip mosaic nodes form what might be regarded as a fine grain homogeneous machine. As mentioned in the introduction, the mosaic node is structurally identical to the cosmic node, with the groundrule that it fit on one MOSIS chip. The full design [Lutz84], with processor, storage, and channels, fits on a chip 6 mm on a side in 3 micron nMOS technology. The complexity of this chip, 4000 λ by 4000 λ, or 16 Mλ^2, and our use of a 3- rather than 1-transistor dynamic RAM cell, allows us only 4K bytes of storage.

While a sample of two is too small to draw any real conclusions, mosaic is at least an example supporting the supposition that smaller nodes can be expected to outperform larger nodes in instruction rate. The mosaic node is so much faster than the cosmic node that its subroutine 32-bit mantissa floating point multiply or add is faster than the 23 bit mantissa floating point in the 8087 floating point coprocessor. Although mosaic includes an integer multiply, it is not otherwise specialized for floating point computations, and its instruction rate for more mundane operations, such as MOVE between a 16-bit register and storage, is about 2.5 Mips. The mosaic processor makes 10 storage references per microsecond, typically 2 for refresh, 5 for instructions and data, and 3 for null references or prefetches that are later discarded.

With only 1/32nd the storage size of the cosmic node, the programming style for mosaic is necessarily "very careful". In particular, these nodes are too small to accommodate much in the way of an operating system. The node is too small for some of the computations currently performed on cosmic cube, unless one breaks up large processes to run (sequentially, if necessary) on groups of mosaic nodes. However, mosaic appears to be a nearly optimal grain size for many simpler problems, including the matrix and graph problems investigated by Sally Browning [Browning80], many-body and finite element computations, signal and image processing, graphics, and circuit simulation.

Mosaic nodes can be connected in many regular communication plans while still a fixed degree (≤ 4) graph. Mosaic was originally intended for a tree connection. However, it is also well suited for

cube connected cycle, shuffle, 2-dimensional mesh, and chordal ring connections. A new "memory mapped" communication section is being designed for mosaic that will allow us to incorporate more ports, and also makes the ports self-timed rather than synchronous.

Although small mosaic systems have been assembled for testing and for programming experiments, the system building project is still in its early stages. The design of the mosaic node is described fully in [Lutz84]; the following descriptions will accordingly be brief.

4.1 Cost/performance of Mosaic Hardware

We are planning to complete a 1024 node mosaic system, configured as a shuffle or mesh, during 1985. This system will likely be packaged as 16 PCBs with 64 nodes per board, an assembly essentially similar in regularity, power distribution requirements, and timing, to the dynamic RAM primary storage used in a large mainframe. Mosaic will also use an intermediate host that is a simple variant on that used for cosmic cube.

Suppose that such systems were mass produced in the style of add-in storage for mainframes, which currently sells for somewhat less than $20 per 64K RAM chip, packaged and powered. The present mosaic node is of comparable complexity and function to a 64K RAM. Even taking a selling price of $50 per chip, to be conservative, a mosaic system with 1024 nodes, just 1024 chips plus clock drivers and interface, would sell for about $50K (exclusive of software).

Analysis of a number of sample regular problems indicate that a mosaic system of this size should be capable of sustained floating point speeds in the same 20-80 million floating point operations per second range that is quoted for the Cray-1. The Cray-1 is superior to most mainframes in the measure of floating point operations per second per dollar cost, and the cost is $5-10M.

Of course there are numerous points on which such a comparison between mosaic and existing supercomputers is unfair, if not invalid, including (1) differences in generality and programmability, (2) differences in primary storage size, (3) software support, and (4) existence. However the ratio of cost/performance in the order of 100 in comparing mosaic with a conventional mainframe or supercomputer would certainly make these machines attractive for the regular applications for which they are most efficient. Similar arguments apply to many other fine grain concurrent systems.

4.2 Applications and Programming Environment

Early work in programming systems and applications for a programmable tree machine by Sally Browning [Browning80, Browning&Seitz81] used a derivative of Hoare's Communicating Sequential Processes (CSP) programming notation for representing numerical and graph algorithms. Peggy Li and Lennart Johnsson [Li83] later developed fast loading algorithms and a prototype operating system [Li81]. In addition, there have been many algorithms published for computational or systolic arrays that map very easily onto a suitably connected mosaic. This work well represents one style of programming mosaic systems, somewhat analogous to the crystalline programming style used in the cosmic cube. However, we are also studying ways of programming mosaic systems derived from experience in the cosmic cube project, in particular, systems supporting message routing and queueing with multiple processes per node.

Since we lack storage space for much of a run time system in the nodes, the host tools for producing mosaic programs are expected to be more sophisticated than those for cosmic cube. There are many ways in which advanced compilers can assist the programmer. For example, in static assignment of processes to nodes, one can combine processes in a way that relieves the run time system from scheduling. Also, a computation that is specified in processes too large to fit into a

single mosaic node can be subdivided to be run on a group of nodes. Whatever concurrencies could be extracted automatically from the process code might be exploited in execution, but even if there were none, the communication necessary in the group to pass control for the single process could be derived by such a compiler.

5. SUPER MESH EXPERIMENT

Guided largely by seeing how our physics colleagues have been programming certain regular applications on the cosmic cube, we started in 1982 investigating the design of an SIMD machine of quite small grain size, about one quarter of size of the mosaic node. Since this project is in its earliest stages, we will only outline some of our reasoning about its design.

It seemed very wasteful in cosmic cube that the same program was loaded into all of the nodes of this MIMD machine, even though the execution temporarily follows different branches in different machines. We have always rationalized this duplication of code as necessary in face of the communication cost of broadcast control. However, if an efficient broadcast mechanism could be devised, an SIMD machine of quite small node size could achieve extraordinary cost/performance characteristics. One might think of this machine as a VLSI incarnation of the Illiac IV.

The approach taken to broadcast control, as well as to provide optimal performance per area in the arithmetic, was to exploit serialism. We use parallel arithmetic in conventional computers largely to balance arithmetic speed against storage cycles, not because it is efficient. Serial addition in systems with very well balanced combinational delays is nearly as fast as parallel addition, and uses only $1/nth$ the logic for n-bit words. Carry- save serial multiplication has a similar advantage over a multiplier array or add-and-shift multiplier. There is no correspondingly efficient bit serial algorithm known for division, but it can be performed through multiplications by Newton's method. Slower serial nodes also simplify the instruction issuing computer and the broadcast of control, indeed allow and encourage this broadcast to occur serially. In somewhat different terms, a way to get the most performance per chip, in a given or scaling technology, is to minimize the area-time product (cost/performance) in the elementary operations, as is done with serial arithmetic, and then to rely on the concurrency of these operations to multiply the performance.

The VLSI engine that evolved from this reasoning, an SIMD array machine dubbed super mesh, resembles a computational or systolic array, somewhat as H T Kung and C L Leiserson described them in [Mead&Conway80], except that instead of the algorithm being built into the node, the node operates on the contents of its registers in response to the instruction sequence broadcast through the ensemble. Given the existing body of algorithms developed for computational arrays, and the programmability and efficiency of this implementation, we are confident that super-mesh will be an interesting and useful VLSI system.

A strawman design of the node includes 64 registers for 64-bit floating point numbers, and a carry-save technique for the mantissa multiplication. Four nodes fit on one rather large chip at 3 micron feature size. Because of the uniformly short paths through combinational logic, such a machine should be able to operate at a bit rate in excess of 20 MHz, which for a 56-bit mantissa translates into about 3 microseconds (65 clock cycles) per floating point multiply. Thus we expect this chip to achieve over one million 64-bit floating point operations per second, in addition to its storage and communication functions. The design allows operands to be fetched in the register

storage and communication to take place concurrently with floating point operations.

This strawman machine turns out to have too much arithmetic performance and too few registers to be a good fit to real problems. It reveals very well a problem inherent in the computational array model [Kung82] that for 2-dimensional problems the array is so fast, $O(n)$ for many problems [Kung80], that the $O(n^2)$ time complexity of serial loading typically exceeds that of computing. A realistic design will have to include much more storage relative to the arithmetic capability, in effect to share the arithmetic capability amongst more operands, and to permit a broader class of applications. A preferred way to increase the storage is not to increase the number of internal registers, but to provide another (hierarchic) storage level of dynamic RAM chips. Details of the control of this secondary store are currently being worked out; suffice it to say that it fits the array model very easily.

We have been tediously careful in the physical design of this machine to be sure that it can be scaled to arbitrary size, as long as a pair of neighboring nodes can be included in a single equipotential region [Seitz80]. (Indeed, this project is in part an experiment in physical synchronization techniques.) It is the physical structure of the clock and microinstruction distribution that requires that the node interconnection plan be a 1-, 2-, or 3-dimensional mesh, which can be mapped onto a manifold such as a torus.

The physical design of super mesh starts with clock distribution, which is accomplished by a 1-, 2-, or 3-dimensional tesselation of locally coupled self-timed circuits, each containing a Muller C-element and a reference delay. The clock skew between neighbors in the network is bounded to a small fraction of the clock period and is therefore arbitrarily large across a diameter of an arbitrarily large machine. The skew between neighbors is *not* unbounded as Fisher and Kung correctly assert would occur [Fisher83] if one used only amplifiers, an approach that would in any case violate the principle that one cannot communicate a time interval across a spatially unbounded structure, even in one dimension. Instead, each clock source contains its own reference interval, and the tesselation network behaves as an enormous set of coupled ring oscillators, with a period determined by the maximum reference delay, and with small local differences in phase that are consistently positive with respect to a specific "corner". The local skew in communication between neighbors is accommodated by the t_{21} time [Seitz80] in two phase clocking being larger than the skew.

Broadcast of the microinstruction is serial, starting from the corner mentioned above, but is resynchronized on each step. Thus the whole array is pipelined, with nodes more distant from the corner being progressively more out of phase. This approach is conceptually identical, but carried out on a bit-by-bit basis, to the transformation of broadcast into pipelining that has been described [Cohen79] for other computational array structures. This pipelining is completely invisible to the programmer.

The design and simulation of the super node has been accomplished by Wen-King Su [Su xxxx:TR:84], hence the name SUper mesh.

The programming system visualized for super mesh is entirely different from those of the MIMD machines. Super mesh operates by microinstruction sequences being transmitted to the array, and can be programmed by specifying the node computation in an ordinary sequential programming notation. Actually, to deal with conditionals and boundary conditions, there are several alternative instruction streams transmitted through the array in parallel. The programming of the control computer, and the ability in serial broadcast to include several instruction streams, leaves open the possibility of compiling the process specification into microprogram sequences for one node or groups

of several nodes. The use of physical submeshes is the same recourse in super mesh as for mosaic for a problem requiring more storage than is provided in a single physical node, and allows also some concurrency in the computation performed per process, a second level of concurrent decomposition that could be extracted in compilation.

6. SUMMARY

With the usual caveat that performance is achieved at the expense of specialization, we are optimistic about our experiments with the programming and engineering of this homogeneous ensemble style of machines, for two reasons:

First of all, these concurrent machines have not proved to be difficult to program by the methods we have described, in which the programmer formulates a computation in terms of concurrent processes. We share the skepticism of others, that one cannot depend on the programmer to write "ultraconcurrent" programs with many different processes with complex interactions. The applications that we represent as reasonable are those in which the computation is demanding because of the number of processes, and in which the degree of concurrency grows with the problem size. Many important and currently intractable computational problems fit this description.

Second, experience with the design and engineering of these machines indicates that they provide for these demanding problems excellent cost/performance characteristics. These machines can exploit the scaling of feature size in VLSI technology, and be scaled to very large numbers of nodes, to provide present supercomputer performance in very small packages, and capabilities well beyond present supercomputers in systems the size and cost of present mainframes.

These experiments may appear to be motivated in largest part by the "greed for speed," the simple desire to use VLSI to achieve very high performance computation. Performance has indeed been an objective that has made these system building experiments excellent vehicles for developing engineering and programming techniques for concurrent VLSI systems, and has provided a concrete way in which we can measure our success or failure. However, we do not see the engineering and programming as being either fundamentally difficult or an end in itself. Rather, we see these system building experiments as exposing the opportunities and thus stimulating research on the more fundamental problems in concurrent computation.

7. ACKNOWLEDGEMENTS

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Projects such as those described have become feasible in a university setting largely because of the MOS implementation system (MOSIS) that is maintained and constantly improved by people at USC/ISI, principally under ARPA support. Special thanks are due the whole MOSIS crew, but particularly to Danny Cohen, George Lewicki, Lee Richardson, Paul Losleben, Kathie Fry, Joel Goldberg, Ron Ayres, and Vance Tyree, for the many "special cases" on which we have received special help.

A contribution by Intel Corporation of chips for the cosmic cube is gratefully acknowledged.

These projects involve extensive collaborations at Caltech. I have tried to indicate in the text the work done by numerous Caltech students over the past several years. The intellectual contributions of several Caltech faculty: Alain Martin, Randy Bryant, Lennart Johnsson (now at Yale), and Martin Rem in computer science, and Geoffrey Fox in high energy physics, should be mentioned as well.

The author very much appreciates the extensive and constructive review of this paper by editors Danny Cohen and H T Kung, and reserves for himself the blame for the large residue of arm-waving statements that even these editors could not entirely expunge.

8. REFERENCES

< in another file, in preparation >

APPENDIX B

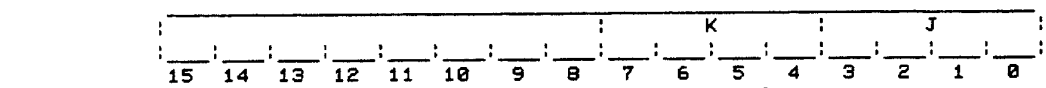
</lutz.tre>instr3.txt

MOSAIC PROCESSOR INSTRUCTION SET (version 3; 26-FEB-84)

PROCESSOR FEATURES:

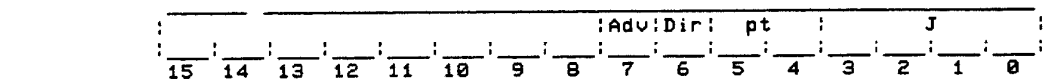
Sixteen 16-bit general registers: R0 ... R15
Memory addressed as 16-bit words
12-bit Program Counter (PC) contains address of next instruction
Flags: C -- Carry/Not Borrow
Z -- Zero
N -- Negative
V -- Twos-complement overflow
Ports: Four input ports
Four output ports
Connecting an input port to an output port forms a fifo
two 16-bit words long.

ALL INSTRUCTIONS:



followed by 0, 1, or 2 words of immediate value.

When K specifies a port:



pt is the port number; specifies one of 4 ports
Dir=0 for output port; Dir=1 for input port
Adv=1 to advance port (remove word from fifo) after input port is read

KEY: Rn is register number n.
Rn++ is register number n, incremented after reading.
--Rn is register number n, decremented before reading.
val is the an immediate value.
@z is the memory word whose address is z.
A : B means concatenation of bit field A and bit field B .
f<i> means the i-th bit of f . f<i:j> means i-th to j-th bits of f .
Bits are numbered from least to most significant.
FPC is the flag/PC word: C : V : N : Z : PC

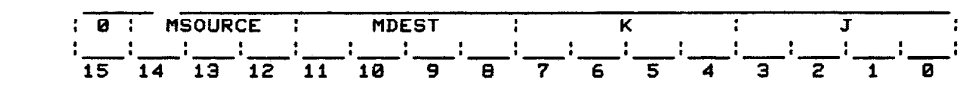
SPECIAL CASES:

HARD RESET: (Reset pin goes HIGH)
0 -> PC

SOFT RESET: (Interrupt pin goes HIGH for >=26 microcycles)
FPC -> @(-1); 0 -> PC

INTERRUPT: (Interrupt pin goes HIGH for 1 microcycle)
FPC -> @(-2); @(-3) -> PC

MOVE INSTRUCTIONS:



Assembly syntax: MOVE <source>,<destination>
Execution time = 3 + Time(MSOURCE) + Time(MDEST) microcycles
Ri means Rk when MSOURCE is 0, 1, 2, or 3; Ri means Rj otherwise.

MSOURCE	X	<source>	Time(MSOURCE)
0	Rj	Rj	0
1	@Rj	@Rj	2
2	@Rj++	@Rj++	2
3	@(Rj+val)	@Rj+val	3
4	val	#val	0
5	@val	@#val	2
6	Input Port pt	Ppt+ [advance]	1
		Ppt= [no advance]	1
7	0	0	0

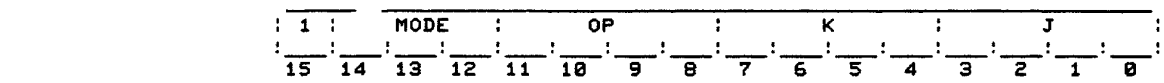
MDEST	effect	<destination>	Time(MDEST)	
0	X -> Ri; set Z,N,V	Ri	0	
1	X -> @Ri; set Z,N,V	@Ri	2	
2	X -> @Ri++; set Z,N,V	@Ri++	2	
3	X -> @(Ri+val); set Z,N,V	@Ri+val	4	
4	X -> @--Ri; set Z,N,V	@--Ri	3	
5	X -> @val; set Z,N,V	@#val	3	
6	X -> Output Port pt; set Z,N,V	Ppt	0	
7	FPC -> @--Ri; X -> PC	PC,@--Ri	4	
8	X -> PC	PC	1	
9	X -> FPC	FPC	1	
A	X -> PC if flag condition k true	PCFT,k	\	
B	X -> PC if flag condition k false	PCFF,k	\	1 if branch is
C	X -> PC if port k is not ready	PCPNR,k	/	taken
D	X -> PC if port k is ready	PCPR,k	/	0 otherwise
E	undefined			
F	undefined			

Notes: When source and destination both contain a val, then they are different (i.e. the instruction takes two immediate values).
"set Z,N,V" means modify Z and N based on value of X; set V to zero.
When source or destination is a port, instruction does not terminate until port is ready (i.e. until input port has a word to read or output port has room for another word).
When source is a port, destination may not be a conditional branch (MDEST= A, B, C, or D) due to contention for field k.

FLAG CONDITIONS:

K	flag condition		K	flag condition	
0	V	[overflow]	4	Z	[zero]
1	N	[negative]	5	Z or N	[<= zero]
2	~C	[Carry = 0]	6	Z or ~C	[unsigned <=]
3	N xor V	[signed <]	7	Z or (N xor V)	[signed <=]

ARITHMETIC INSTRUCTIONS:



Execution time = 3 + Time(MODE) + (IF MULTIPLY THEN 18) microcycles

MODE	X	Y	Dest	Assembly language syntax (when Rk not specified, k=j)	Time(MODE)
0	Rj	Rk		<OP Mnemonic> Rj {, Rk}	0
1	val	Rj		<OP Mnemonic> #val , Rj	0
2	Rj	Rk	Rk	<OP Mnemonic> Rj {, Rk}	0
3	val	Rj	Rk	<OP Mnemonic> #val , Rj {, Rk}	0
4	@Rj	Rk	Rk	<OP Mnemonic> @Rj {, Rk}	2
5	@val	Rj	Rk	<OP Mnemonic> @#val , Rj {, Rk}	2
6	@Rj	Rk	@Rj	<OP Mnemonic> M @Rj {, Rk}	4
7	@val	Rj	@val	<OP Mnemonic> M @#val {, Rj}	4

All Arithmetic instructions modify the Z, N, and V flags.
Instructions which don't do twos-complement arithmetic always set V to zero (MUL, BITT, ASR, ROR, LSR, RNR, AND, OR, XOR, and COM).

MODE = 0 or 1:

OP	Instruction	<OP Mnemonic>	Effect	Carry flag modified?
0	unsigned MULtiply	MUL	high word(X*Y) -> RJ low word(X*Y) -> RK modify Z, N, V based on high word	no
1	CoMPare	CMP	X - Y	yes
2	BIT Test	BITT	X and Y	no
3...F	undefined			

MODE = 2, 3, 4, 5, 6, or 7:

OP	Instruction	<OP Mnemonic>	Effect		Carry flag modified?
0	INCrement	INC	X + 1	-> Dest	no
1	DECrement	DEC	X - 1	-> Dest	no
2	Arithmetic Shift Right	ASR	X<15>:X	-> Dest:C	yes
3	Arithmetic Shift Left	ASL	X + X	-> Dest	yes
4	ROtate Right	ROR	C:X	-> Dest:C	yes
5	ROtate Left	ROL	X + X + C	-> Dest	yes
6	Logical Shift Right	LSR	0:X	-> Dest:C	yes
7	Rotate Nibble Right	RNR	X<3:0> : X<15:4>	-> Dest	no
8	ADD	ADD	X + Y	-> Dest	yes
9	ADD with Carry	ADDC	X + Y + C	-> Dest	yes
A	SUBtract	SUB	Y - X	-> Dest	yes
B	SUBtract and Negate	SUBN	X - Y	-> Dest	yes
C	bitwise AND	AND	X and Y	-> Dest	no
D	bitwise OR	OR	X or Y	-> Dest	no
E	bitwise eXclusive OR	XOR	X exclusive or Y	-> Dest	no
F	bitwise COMplement	COM	~X	-> Dest	no

STANDARD ASSEMBLY MACROS:

Macro call	Definition	Macro call	Definition
JUMP addr	MOVE addr,PC	JRST addr	MOVE addr,FPC
CALL addr,reg	MOVE addr,PC,@--reg	RETURN reg	MOVE @reg++,PC
PUSH arg,reg	MOVE arg,@--reg	POP reg,arg	MOVE @reg++,arg
BUS addr	MOVE addr,PCFT,0	BUC addr	MOVE addr,PCFF,0
BNS addr	MOVE addr,PCFT,1	BNC addr	MOVE addr,PCFF,1
BCC addr	MOVE addr,PCFT,2	BCS addr	MOVE addr,PCFF,2
BLT addr	MOVE addr,PCFT,3	BGE addr	MOVE addr,PCFF,3
BEQ addr	MOVE addr,PCFT,4	BNE addr	MOVE addr,PCFF,4
BLEZ addr	MOVE addr,PCFT,5	BGTZ addr	MOVE addr,PCFF,5
BLOS addr	MOVE addr,PCFT,6	BHI addr	MOVE addr,PCFF,6
BLE addr	MOVE addr,PCFT,7	BGT addr	MOVE addr,PCFF,7
BONR addr,pt	MOVE addr,PCPNR,pt	BOR addr,pt	MOVE addr,PCPR,pt
BINR addr,pt	MOVE addr,PCPNR,pt+4	BIR addr,pt	MOVE addr,PCPR,pt+4

```
<lutz.tree> vrodr 3.txt
! UCODE3.TXT:  MOSAIC PROCESSOR MICROCODE parsed by chip3.sim
! Implements instruction set with complex moves, multiply,
!   and external interrupt.
! Arithmetic modes include two special modes for mul, cmp, and bitt.

  Version 25-FEB-84

! Syntax of implicants is:
! word  <inputs>  ::  <outputs>

! 'I=' starts instruction register mask starting with I<15>.
! Unspecified bits are '*' (don't care).
! When 'I=' not specified for an implicant, that from last implicant is used.


! MACROS:
DEF  Cin=0        Cforce
DEF  Cin=1        Cforce Cval1
DEF  PC++->A      PC->inc Add1 inc->A A->PC
DEF  RA++->A      RA->inc Add1 inc->A A->RA
DEF  PC->A        PC->inc inc->A A->PC
DEF  RJ=>         useJ R=>
DEF  RK=>         R=>
DEF  RJ           useJ R
DEF  RK           R
DEF  X=>          GP= 0C Cin=0 nosh W=>
DEF  Y=>          GP= 0A Cin=0 nosh W=>
DEF  X+Y=>        GP= 86 Cin=0 nosh W=>
DEF  Y+1=>        GP= 0A Cin=1 nosh W=>
DEF  Y-1=>        GP= A5 Cin=0 nosh W=>
DEF  X+1=>        GP= 0C Cin=1 nosh W=>
DEF  X-1=>        GP= C3 Cin=0 nosh W=>
DEF  -1=>         GP= 0F Cin=0 nosh W=>
DEF  0=>          GP= 00 Cin=0 nosh W=>
    ! rnib  refreshes the carry flag.  This is done on the cycle
    ! after DISPATCH: of every instruction.
DEF  saveC        rnib
DEF  testX        GP= 0C Cin=0 nosh setZNV


! FEEDBACK MNEMONICS:
DEF  .reset2         FB= 0 0 0 0 0
DEF  .reset3         FB= 0 0 0 0 1
DEF  .reset4         FB= 0 0 0 1 0
DEF  .interrupt2     FB= 0 0 0 1 1
DEF  .interrupt3     FB= 0 0 1 0 0
DEF  .interrupt4     FB= 0 0 1 0 1
DEF  .interrupt5     FB= 0 0 1 1 0
DEF  .interrupt6     FB= 0 0 1 1 1
DEF  .refetch        FB= 0 1 0 0 0
DEF  .fetch          FB= 0 1 0 0 1
DEF  .decode         FB= 0 1 0 1 0
DEF  .get            FB= 0 1 1 0 0
DEF  .get2           FB= 0 1 1 0 1
DEF  .get3           FB= 0 1 1 1 0
DEF  .get4           FB= 0 1 1 1 1
DEF  .go             FB= 1 0 0 0 0
DEF  .go2            FB= 1 0 0 0 1
DEF  .go3            FB= 1 0 0 1 0
DEF  .go4            FB= 1 0 0 1 1
DEF  .arith          FB= 1 0 1 0 0
DEF  .store          FB= 1 0 1 0 1
DEF  .RJ++--         FB= 1 0 1 1 0
DEF  .PC-2           FB= 1 0 1 1 1
```

```
! HARD RESET

word hardreset:  RESET=1  FB= *  I= *     ::  INT:=0  X Y  RA++->A  .reset4

! INTERRUPT AND SOFT RESET

  ! Interrupt:  PCF-1 -> @(-1);  @(-2) -> PC.
word interrupt1: .decode      I= * INT=1  ::  INT:=0   PC=> X       .interrupt2
word interrupt2: .interrupt2              ::           X-1=> D       .interrupt3
word interrupt3: .interrupt3              ::            -1=> A X     .interrupt4
word interrupt4: .interrupt4       INT=0  ::  Write    X-1=> A       .interrupt5
word interrupt5: .interrupt5              ::                        .interrupt6
word interrupt6: .interrupt6              ::     IN=> A A->PC       .fetch

  ! If interrupt persists, do a soft reset:  PCF-1 -> @(-3);  0 -> PC.
word softreset:  .interrupt4       INT=1  ::           X-1=> X       .reset2
word reset2:     .reset2                  ::           X-1=> A       .reset3
word reset3:     .reset3                  ::  Write                  .reset4
  ! While waiting for INT=0, keep storage static and X and Y digital:
word reset4:     .reset4           INT=1  ::  INT:=0  X Y  RA++->A  .reset4
word reset4:     .reset4           INT=0  ::       0=> A A->PC       .fetch


! FETCH AND DECODE

word fetch:    .fetch  I= *       ::                          PC++->A  .decode
       ! All instructions pass through  decode:  (or  interrupt1: )
word decode:   .decode I= * INT=0 :: IN->I  saveC  RJ=> X Y D M  RA++->A .get
```

```
! REFETCH ON FAILED I/O IN MOVES

word refetch:  .refetch   I= *          :: X-1=> A  A->PC                .fetch


! MOVE SOURCES USING REGISTER J (SO DESTINATION USES K)

word RJ->:       .get       I= 0 0 0 0    ::  saveC              PC->A    .go

word @RJ->:      .get       I= 0 0 0 1    ::  saveC   RJ=> A             .get3

word @RJ++->:    .get       I= 0 0 1 0    ::  saveC   RJ=> A Y           .get2
word @RJ++->2:   .get2                     ::          Y+1=> RJ   PC->A   .get4

word @(RJ+#)->:  .get       I= 0 0 1 1    ::  saveC   IN=> Y    PC++->A  .get2
word @(RJ+#)->2: .get2                     ::          X+Y=> A            .get3

word @-wait:   .get3        I= 0 * * *    ::                     PC->A    .get4
word any-@:    .get4        I= 0 * * *    ::           IN=> X D          .go


! MOVE SOURCES NOT USING REGISTER J (SO DESTINATION USES J)

word #->:        .get        I= 0 1 0 0    ::  saveC   IN=> X D  PC++->A .go

word @#->:       .get        I= 0 1 0 1    ::  saveC   IN=> A            .get2
word @#->2:      .get2                     ::                  PC++->A  .get4

word badPt->:  .get          I= 0 1 1 0 * * * * * 0 :: saveC   PC=> X  .refetch
    ! Wait a cycle so controller can dispatch on port condition:
word InPt->:     .get        I= 0 1 1 0 * * * * * 1 :: saveC       PC->A .get2
word InPt->:     .get2 PortC=0 I= 0 1 1 0 * * * * 0 1 :: Pt=> X D          .go
word InPtA->:  .get2 PortC=0 I= 0 1 1 0 * * * * 1 1 :: Pt=> X D  Advance .go
word iofailed: .get2 PortC=1 I= 0 1 1 0             :: PC=> X           .refetch

word 0->:      .get         I= 0 1 1 1    ::  saveC    0=> X D  PC->A   .go
```

```
! STUFF ASSOCIATED WITH MOVE DESTINATIONS

   ! Use register J in next two cycles only if MOVE source didn't use it:
word pickJ:      .go      I= 0 1 * * * * * *  :: NOTRANSISTORS  useJ
word pickJ2:     .go2     I= 0 1 * * * * * *  :: NOTRANSISTORS  useJ

word store:      .store   I= 0                :: WRITE      PC->A        .fetch


! MOVE DESTINATIONS PROPER

word ->R:        .go      I= 0 * * * 0 0 0 0  ::  X=> R  setZNV  PC++->A  .decode
word ->@R:       .go      I= 0 * * * 0 0 0 1  ::  testX  R=> A            .store

word ->@R++:     .go      I= 0 * * * 0 0 1 0  ::  testX  R=> A X          .go2
word ->@R++2:    .go2                         ::  Write  X+1=> R  PC->A   .fetch

word ->@(R+#):   .go      I= 0 * * * 0 0 1 1  ::  testX      R=> X        .go2
word ->@(R+#)2:  .go2                         ::   IN=> Y     PC++->A      .go3
word ->@(R+#)3:  .go3                         ::  X+Y=> A                  .store

word ->@--R:     .go      I= 0 * * * 0 1 0 0  ::  testX      R=> X        .go2
word ->@--R2:    .go2                         ::  X-1=> A R                .store

    ! Note: Wait a cycle before taking #value from IN to guarantee correct data
    ! independent of MOVE source:
word ->@#:       .go      I= 0 * * * 0 1 0 1  ::  testX          PC++->A   .go2
word ->@#2:      .go2                         ::  IN=> A                   .store

   ! MOVE to output port: if K specifies input port, then do nothing;
   !                      if port isn't ready, reverse side effects and refetch
word ->Pt:    PortC=0 .go I= 0 * * * 0 1 1 0 * 0 :: X=> Pt setZNV PC++->A .decode
word ->badPt:         .go I= 0 * * * 0 1 1 0 * 1 ::        setZNV PC++->A .decode
word ->Ptwait:  PortC=1 .go I= 0 0 0 * 0 1 1 0  ::  PC=> X             .refetch
word ->Ptwait:  PortC=1 .go I= 0 0 1 0 0 1 1 0  ::   Y=> RJ            .RJ++--
word RJ++--:  .RJ++--                           ::  PC=> X             .refetch
word ->Ptwait:  PortC=1 .go I= 0 0 1 1 0 1 1 0  ::  PC=> X             .PC-2
word ->Ptwait:  PortC=1 .go I= 0 1 0 * 0 1 1 0  ::  PC=> X             .PC-2
word PC-2:     .PC-2        I= *                :: X-1=> X             .refetch
word ->Ptwait:  PortC=1 .go I= 0 1 1 * 0 1 1 0  ::  PC=> X             .refetch

word ->PushJ:    .go      I= 0 * * * 0 1 1 1  ::    R=> Y                 .go2
word ->PushJ2:   .go2                         ::  Y-1=> R A               .go3
word ->PushJ3:   .go3                         ::   PC=> D                 .go4
word ->PushJ4:   .go4                         ::  Write  X=> A A->PC      .fetch

word ->PC        .go      I= 0 * * * 1 0 0 0  ::  X=>   A A->PC           .fetch
word ->FPC       .go      I= 0 * * * 1 0 0 1  ::  X=> F A A->PC           .fetch

word /->PC(F=0): .go  FlagC=0  I= 0 * * * 1 0 1 0  ::  PC++->A        .decode
word ->PC(F=1):  .go  FlagC=1  I= 0 * * * 1 0 1 0  ::  X=> A  A->PC   .fetch
word /->PC(F=1): .go  FlagC=1  I= 0 * * * 1 0 1 1  ::  PC++->A        .decode
word ->PC(F=0):  .go  FlagC=0  I= 0 * * * 1 0 1 1  ::  X=> A  A->PC   .fetch
word /->PC(P=0): .go  PortC=0  I= 0 * * * 1 1 0 0  ::  PC++->A        .decode
word ->PC(P=1):  .go  PortC=1  I= 0 * * * 1 1 0 0  ::  X=> A  A->PC   .fetch
word /->PC(P=1): .go  PortC=1  I= 0 * * * 1 1 0 1  ::  PC++->A        .decode
word ->PC(P=0):  .go  PortC=0  I= 0 * * * 1 1 0 1  ::  X=> A  A->PC   .fetch

    ! Undefined destinations
word ->???:      .go          I= 0 * * * 1 1 1 *  ::  testX  PC++->A .decode
```

```
! ARITHMETIC SOURCES

word J,K:       .get   I= 1 0 0 0     ::  saveC  PC->A     RK=> Y M     .go
word #,J:       .get   I= 1 0 0 1     ::  saveC  PC++->A   IN=> X       .go

word J,K,K:     .get   I= 1 0 1 0     ::  saveC  PC->A     RK=> Y       .arith
word #,J,K:     .get   I= 1 0 1 1     ::  saveC  PC++->A   IN=> X       .arith

word @J,K,:     .get   I= 1 1 * 0     ::  saveC            RJ=> A       .get2
word @J,K,K:    .get2  I= 1 1 0 0     ::         PC->A     RK=> Y       .get3
word @J,K,@J:   .get2  I= 1 1 1 0     ::                   RK=> Y       .get3

word @#,J,:     .get   I= 1 1 * 1     ::  saveC            IN=> A       .get2
word @#,J,K:    .get2  I= 1 1 0 1     ::         PC++->A                 .get3
word @#,J,@#:   .get2  I= 1 1 1 1     ::                                 .get3

word any@:      .get3  I= 1 1 * *     ::                   IN=> X       .arith
```

```
! WORDS TO SPECIFY ALU FUNCTION IN NORMAL ARITHMETICS

word inc:  .arith I= 1 * * * 0 0 0 0  ::  ALUONLY GP= 0C Cin=1 nosh setZNV
word dec:  .arith I= 1 * * * 0 0 0 1  ::  ALUONLY GP= C3 Cin=0 nosh setZNV
word asr:  .arith I= 1 * * * 0 0 1 0  ::  ALUONLY GP= 0C Cin=0 asr  setZNV
word asl:  .arith I= 1 * * * 0 0 1 1  ::  ALUONLY GP= C0 Cin=0 nosh setZNV setC
word ror:  .arith I= 1 * * * 0 1 0 0  ::  ALUONLY GP= 0C Cin=0 ror  setZNV
word rol:  .arith I= 1 * * * 0 1 0 1  ::  ALUONLY GP= C0       nosh setZNV setC
word lsr:  .arith I= 1 * * * 0 1 1 0  ::  ALUONLY GP= 0C Cin=0 lsr  setZNV
word rnr:  .arith I= 1 * * * 0 1 1 1  ::  ALUONLY GP= 0C Cin=0 rnib setZNV
word add:  .arith I= 1 * * * 1 0 0 0  ::  ALUONLY GP= 86 Cin=0 nosh setZNV setC
word addc: .arith I= 1 * * * 1 0 0 1  ::  ALUONLY GP= 86       nosh setZNV setC
word sub:  .arith I= 1 * * * 1 0 1 0  ::  ALUONLY GP= 29 Cin=1 nosh setZNV setC
word subn: .arith I= 1 * * * 1 0 1 1  ::  ALUONLY GP= 49 Cin=1 nosh setZNV setC
word and:  .arith I= 1 * * * 1 1 0 0  ::  ALUONLY GP= 08 Cin=0 nosh setZNV
word or:   .arith I= 1 * * * 1 1 0 1  ::  ALUONLY GP= 0E Cin=0 nosh setZNV
word xor:  .arith I= 1 * * * 1 1 1 0  ::  ALUONLY GP= 06 Cin=0 nosh setZNV
word com:  .arith I= 1 * * * 1 1 1 1  ::  ALUONLY GP= 03 Cin=0 nosh setZNV


! NORMAL ARITHMETIC DESTINATIONS

word ALU->K:   .arith I= 1 0 1 *      ::  PC++->A  NOALU  W=> RK    .decode
word ALU->K:   .arith I= 1 1 0 *      ::  PC++->A  NOALU  W=> RK    .decode

word ALU->@:   .arith I= 1 1 1 *      ::           NOALU  W=> D     .store
word ALU->@J:  .store I= 1 1 1 0      ::  Write  PC->A                .fetch
word ALU->@#:  .store I= 1 1 1 1      ::  Write  PC++->A              .fetch


! SPECIAL ARITHMETICS:  MULTIPLY, ETC.

word mul:  .go  I= 1 0 0 * * * 0 0    :: Mshift  0=> Y  SRin=1          .go2
word mul0:    .go2  SRout=0  Mout=0   :: Mshift    Y=> Yshift  RA++->A  .go2
word mul1:    .go2  SRout=0  Mout=1   :: Mshift  X+Y=> Yshift  RA++->A  .go2
word muldone: .go2  SRout=1           :: Mshift    Y=> RJ setZNV PC->A  .go3
word mulend:  .go3                    ::           M=> RK       PC++->A  .decode

word cmp:  .go  I= 1 0 0 * * * 0 1    :: PC++->A GP= 49 Cin=1 nosh setZNV setC
                                                                  .decode
word bitt: .go  I= 1 0 0 * * * 1 *    :: PC++->A GP= 08 Cin=0 nosh setZNV
                                                                  .decode
```